The Thesis Committee for Yosef Saputra
Certifies that this is the approved version of the following Thesis:

Programming Abstraction for User-Driven Architecture in the Internet of Things

APPROVED BY
SUPERVISING COMMITTEE:

Christine Julien, Supervisor

Edison Thomaz

Programming Abstraction for User-Driven Architecture in the Internet of Things

by

Yosef Saputra

Thesis

Presented to the Faculty of the Graduate School
of The University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2019

I dedicate this thesis to myself, my wife, my parents, my siblings, and the community for a better future.

Acknowledgments

I would like to express my sincere appreciation and gratitude to Dr. Christine Julien for giving me her continued assistance on whatever I did during my study, especially on the research opportunity and the constructive feedback for my research project and my thesis. I would like to thank Dr. Edison Thomaz for his patience and time to provide me with great feedback on my thesis.

Some portions of this thesis have been taken from the author's own published paper, with the permission of the co-authors. The title of the published paper is *"Warble: Programming Abstractions for Personalizing Interactions in the Internet of Things"* (Saputra et al., 2019). Therefore, I would like to express many thanks to the co-authors: Dr. Christine Julien, Nathaniel Wendt, Jie Hua, and Dr. Gruia-Catalin Roman, for providing both technical and non-technical support to me on this research project.

I also would like to thank Dr. Mary Eberlein to give me the opportunity to work as a Teaching Assistant which eventually enables me to attain a deeper understanding of software engineering principles to support my research project.

Last but not least, I would like to thank my wife, my parents, my siblings, and other family members that always deliver their supports in any forms to me to complete this thesis.

Abstract

Programming Abstraction for User-Driven Architecture in the Internet of Things

Yosef Saputra, MSE

The University of Texas at Austin, 2019

Supervisor: Christine Julien

The advancement of smart devices and wireless networking have been enabling the Internet of Things to establish a presence in the market. Manufacturers offer different IoT solutions for a vast range of IoT deployments, starting from home automation to building smart cities. Despite the immense progress on creating the physical building blocks of IoT, there is an essential need to define how to manage the vast deployment of the devices. There will be a tremendous increase of information from the devices. Managing the intelligent devices to provide an intuitive IoT experience requires a software abstraction with a scalable and effective architecture. Through research on the Warble platform, we encapsulate our exploration of the architectural model to resolve the IoT management problem. Enabling interoperability and personalized IoT experiences needs a middleware that embraces a user-driven approach to communicate with assorted devices across multiple manufactures and ecosystems. We introduce the Warble middleware, an IoT management middleware with an extensible abstraction of personalization and interoperability. The middleware abstracts the complexity of communicating across various devices, and enables applications to learn from prior user interactions within the IoT space. Furthermore, we also introduce Mesh, an IoT framework for research and development of IoT model architecture, which enables the creation of IoT and to define their structured collaborations. Through this

thesis, we present the architecture and the internal mechanisms of both software artifacts. Subsequently, we evaluate our implementations through a use case that demonstrates their contributions to IoT research.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Internet of Things changes the way we live, and we will all soon be part of the revolution. The implementation is not limited to improving home automation but also includes industrial, health, and countless more applications. With time, we find more everyday devices have the necessary computational capability and are available wirelessly. Electronic chips are getting smaller and more powerful, becoming readily able to support the device requirements for automation. While smart devices are quickly emerging as the endpoints, the challenge still remains unresolved on the systematic strategy to embed these smart devices into the fabric of human life (Weiser, 1999).

Unconsciously, the nature of most human interactions with devices encompasses three processes, i.e, SENSING the target domain, ANALYZING needs or intentions to change the domain state, and EXECUTING a series of actions to influence the domain. The IoT could generously help to manage interactions that are predictable, trivial, time-consuming, physically unreachable, out-of-scale, or even dangerous to do. Some examples are switching on the appropriate lights in a room (predictable and trivial), monitoring a security camera for suspicious activities all day (time-consuming), locking the front door of a house remotely (physically unreachable), diagnosing vehicle engine health (out-of-scale), sensing possible radioactive leaks in a nuclear power plant (dangerous), and so on.

At present, smart devices are collectively able to do most, if not all, of these processes, enabling the future likelihood for non-automated interactions to be automated. Sensors and actuators could perform collect information about the environment and influence the physical environment respectively whereas computers are able to apply analytics towards the informative input data. Nevertheless, the prime challenge is to unify these functional processes to work together as if they understand each other. From another perspective, the IoT can be seen as a complex data management and manipulation problem.

One of the objectives in this study is to introduce a programming abstraction adjacent to the user, taking a form of a management middleware, called the WARBLE *middleware*. It allows the user to continuously discover surrounding devices

and to provide interaction channels to the most relevant and available devices to embody the intent of the user in physical space. The middleware sits on a personal mobile device to act as a proxy for the user in the space. We employ a user-driven approach, keeping all the sensing and analysis in the user side. The output of the middleware is a series of explicit commands to the devices.

The fact that a vast diversity of devices exists requires WARBLE *middleware* to explicitly enable *interoperability*. At present, there are many different ecosystems introduced by different vendors, for example, Nest (Nest, 2011), Philips Hue (Philips Hue, 2012), Withings (Withings, 2009), Wink (Wink, 2014). Each ecosystem has its own workspace having an imaginary barrier which makes them incompatible with each other, including the core analytic components, the interfaces, and the communication channels. WARBLE embraces the variety of ecosystems rather than defining a platform-independent standard. We presume that diversity is a great asset to converge into the most optimized and mature IOT.

Secondly, human-device interaction is not apart from personal preference. WARBLE *middleware* design incorporates *personalization* by direct user feedback and prior user interaction analysis. It records the user *context* when performing a specific interaction, where *Context* is defined as an instance of environmental and non-environmental situation in a given time and location. This study does not discuss an optimized algorithm to represent the personalization, yet the middleware enables the extensible resources in its form.

Thirdly, the WARBLE *middleware* also promotes *ease of programming* for top-level applications to enter the IOT domain. Developing a user application, which works on a variety of ecosystems and transforms user contexts to personalized device actions, requires significant work and prior knowledge in software engineering, especially in the IOT. Therefore, the middleware administers IOT operations that simplify the work of the programmers.

Lastly, this study also extends to another objective to facilitate further research on modeling the human-device interactions in a true pervasive-computed system. We introduce an IOT framework, MESH. MESH is designed to emulate a 3D space and its surroundings in detail, allowing researchers to have a detailed representation of space, entity, and their interactions. In current MESH, the fabric of space and each of its space factor, like temperature and luminosity, is modeled as a 3-

dimensional array. The vertical collection of elements across the space-factor arrays represents a single unit cube of space representation. Additionally, the resolution of the arrays is modifiable with the expense of computing capability to administer more detailed capability to the IOT system. With these highlighted features, MESH provides a useful platform to research the design strategies to build the IOT technology.

Although all the studies covered this thesis do not cover a comprehensive set of concepts, algorithms, building blocks, models, and implementation approaches to achieve WARBLE's end goal, its contribution is to bring one step closer to realizing the WARBLE vision. The vision starts by simplifying the complex IOT design problem and proposing solutions from different perspectives to resolve the problems step by step.

1.1 INTERNET OF THINGS and its Ubiquity

The concept of emerging IOT is ubiquitous. Current technology has been able to successfully invent an interconnected virtual world using the Internet. This advancement has changed the way we live, work, read, shop, among many other activities. However, this growing trend is limited to the scope where digital information could reach. Inherently, the INTERNET OF THINGS intends to break this barrier and penetrate into the real physical world. The impact of the IOT would broaden the technology scope from traditional computers, like PC and smartphones, to everyday objects. Therefore, IOT needs two important enablers in the form of sensor and actuator, to observe and to transform the tangible space.

At present, many research studies aim at resolving specific implementation problems, starting from the core building components, architecture and design, algorithm, human-device interactions. The ubiquity means that IOT is applicable to extensive deployment levels, ranging from personal wearables, home, industry, city, and the global level. Many studies start from making a specialized assumption in a specific deployment level. Subsequently, upward or downward generalization might be worked out in the study.

Similar to other studies, this study builds on the context of home and neighborhood automation. Aligned with the essence of pervasive computing, WARBLE

observes and disentangles the complexity of human-device interaction. It finds that current IOT implementations are still a ways from the expected IOT. As an example, one home automation uses a voice command to switch on a light. The command is appended with a unique light name reference which was assigned in advance during the setup. This interaction is problematic for a new occupant or visitor who is unfamiliar with the light name. Finding the light name then may be more strenuous than finding the manual light switch, and the purpose of IOT is simply defeated. WARBLE targets an effective solution that utilizes a sensor network to approximate the user presence. Subsequently, the system analytic model triggers a series of actions to illuminate the space near the user. Switching on a light or raising the window blinds are two possible options depending on the time of day. WARBLE's objective is to achieve intuitive interactions rather than presenting a control panel on the mobile device.

Besides the operating area and the types of services, the heterogeneity among different deployment levels lies on the extent of manual configurability. Compared to the deployments in home automation or smart city, there is less predictability in the industry-level deployment. At home settings, most human-device interactions have a direct influence on the users because there are fewer possible concrete interactions. In contrast, the interactions in industry focus to align with the situational industrial strategy. This setting requires a higher degree of data accessibility and device configurability to control the system. Therefore, we expect the IOT industrial deployment to receive more instances of explicit intent from the user. For example, at home, a user is usually uninterested to know how long the light is illuminating the room. However, in industry, it is typical for the operator to know the utilization rate for a piece of equipment in order to formulate an improved operational strategy.

The IOT virtually makes our existing and, potentially, future human knowledge literally everywhere. It opens many opportunities for other technology advancements to permeate the fabric of human life. At present, we are still conscious that we interact with smart devices, such as smartphones and home smart speakers. However, the advancement and ubiquity of the IOT will turn these special interactions into commonly natural interactions.

1.2 Thesis Overview

This section describes the overview and flow of the entire thesis. Chapter 1 starts with the introduction of the INTERNET OF THINGS as a part of ubiquitous computing and how the study in this thesis contributes to its advancement. Chapter 2 explains how this study views the INTERNET OF THINGS and its scope. It also gives an imaginary example of IOT situation that the study has been trying to achieve. Subsequently, we will also see the approach that the study takes to crack an integrated IOT problem into smaller problems.

In this thesis, there are two studies. One study contributes to solving an IOT architectural problem through a middleware whereas the other introduces a software framework to simulate the implementation of the solution for the architectural problem. Chapter 3 contains the related work for both studies. We will go through some researches that provide the state-of-the-art in resolving this problem. Chapter 4 elucidates the middleware, how it works, and the software implementation. Chapter 5 explains the framework objective in more details, the features, and the implementation. Chapter 6 exhibits the evaluation of both studies from the programming perspective. Lastly, Chapter 7 concludes the contribution of the studies.

Chapter 2

WARBLE Vision

This chapter discusses the overview of how the WARBLE vision to support a future IOT. Albeit drilling the technical details, it disintegrates the IOT engineering problem into multiple structured fragments which are more focused and manageable to be resolved.

2.1 WARBLE

The study in this thesis is a part of a larger research project called WARBLE (Saputra et al., 2019)[1][2]. WARBLE studies scientific approaches to realize the INTERNET OF THINGS with an emphasis on natural human-device interactions. WARBLE has been exploring varieties of engineering strategies to model how the entire IOT system should be architected.

2.2 Imagining INTERNET OF THINGS

To improve our understanding of WARBLE's scope and its level of detail in actualizing the IOT, this section illustrates a handful of imaginary IOT situations that WARBLE pursues.

Dom is a professor. Dom has been using a mature IOT *system at home. The system takes care of countless devices in the house. As a result, Dom's preferences are mostly abstracted in Dom's mobile phone and cloud, depending on the needed level of confidentiality. This data is accessible by Dom to control the device behavior when interacting with the* IOT *space. Being well-known for his* IOT *research, Prof. Dom is scheduled to present his research at the University of Atlantis as a guest speaker tomorrow noon. His flight is today.*

[1] Yosef Saputra, Jie Hua, Nathaniel Wendt, Christine Julien, and Gruia-Catalin Roman. Warble: Programming abstractions for personalizing interactions in theinternet of things. MOBILESoft, 6, 2019.

[2] The author contribution includes planning research, performing research, developing research artifacts, analyzing experimental data, improving performance, writing technical papers, and writing the thesis.

As soon as his driverless ride arrives, Dom realizes that he forgot to bring his lucky hat stored in the attic. Therefore, he rushes to the dark attic. However, the light does not turn on automatically when he enters. He has to find the manual switch and looks for his hat. After finding it, he dashes out of his house and starts his trip. He is currently on the ride towards the airport. Thinking why he forgot to equip his attic with IoT *devices, he plans to just install two* IoT *devices, i.e., a light and a motion sensor, in the right place. He does not need to configure anything else, because the house* IoT *system will configure them intelligently.*

Unfortunately, he spills his morning coffee while imagining his plan. Therefore, as soon as he arrives at the airport entrance, he looks for a restroom. First, he checks his mobile phone and, without even unlocking the phone, it displays the map of the airport. The phone understands his present situation. After locating the nearest restroom quickly, he uses one of the bathroom sinks to wash his hands. The running water is at his favorite temperature. However, it is too cold for an unexpectedly chilly morning. Therefore, he adjusts it on the tap lever manually.

His flight went smoothly and he arrives at the hotel. The IoT *is ubiquitous and this hotel is not an exception. When he enters his room, his new living space for the next three days, the lights are automatically turned on to create an ambiance similar to his home. He washes his hands again in the bathroom and the temperature is perfect now despite the cold weather. After the flight rush, he decides to read a book in the room until dinner time. He sits on the couch and opens the book. His wearable detects this activity and works together with the room to create a reading light ambiance. Soon, he forgets that he is away from home.*

*The next day starts very well for him. He does not forget his coffee at the hotel. The coffee machine has it brewed as he likes just before he arrives at the machine. Last night, he opted for his ride to the university to pass several attractions in the city for sightseeing. He did it on his phone. And so, his ride to the university is full of excitements through the sweet spots in the city. After arriving at the university, his acquaintances welcome and accompany him to the auditorium. He is ready to give his talk to a big audience of students. As soon as he begins talking, the light setting in the auditorium is automatically adjusted to presentation mode. This is not his preference, yet it is the common light setting which most presenting speakers prefer. He says "*IoT *it is."*

2.3 INTERNET OF THINGS Building Blocks

The ubiquity of the IOT, as a bridge between the virtual and physical worlds, asks for an easily extensible, scalable, and robust programming abstraction layer to organize the system. This section unravels the IOT building blocks envisioned by WARBLE.

2.3.1 IOT System

First, we start by defining the IOT system. An IOT system is a collection of all components in an IOT space, including properties of the space, physical devices embedded in the space, users, IOT models, and the possible interactions among the devices within the space. Although we can discretize the IOT space explicitly, the space boundary is flexible as we refer it. As an example, we can refer two houses with their shared front yard as an IOT space.

In this study, the space property is abstracted from the entirety of space because it is the playground of IOT devices today. The first-level penetration of virtual technology into the physical world lies in this abstraction. Not only extracting information, the IOT also intends to manipulate it intelligently according to the patterns of observable human interactions. Subsequently, the technological penetration could continue closer to human properties, such as needs, intents, and ideas. As an example, the IOT could understand the user's hunger and thirst. This requires far more advanced technological support to establish the abstractions of these aspects, but they are still within the scope of IOT.

2.3.2 IOT Model

The IOT model is the heart of IOT technology to realize a space that is naturally responsive to the users. The main purposes of an IOT model can be formalized as follows:

- to adapt with the available services either locally or remotely, existing or newly discovered
- to build context awareness actively from the available sensors (SENSING)

- to abstract the human-device interaction patterns from prior data and timely user feedback (ANALYZING)
- to transform a user's natural intent into a series of automated actions to manipulate the IOT space as the user desires (EXECUTING)

A simple concrete example is a "nearest" model which represents a user that always selects the nearest IOT device. In contrast, a complex model may assume an ensemble model of two trained neural networks (Haykin, 1998) that impersonates adult and child occupants. By utilizing one or more appropriate IOT models, the system could intelligently manage most, if not all, of the explicit interactions that are predictable, trivial, time-consuming, physically unreachable, out-of-scale, or dangerous.

There are two main categories of IOT model, i.e., static and adaptive. As the IOT system is a dynamic and evolving environment, the adaptive approach is more reasonable for building an IOT model (Ariza et al., 2018). It allows the model to mutate over time as new patterns develop. The mutation needs an external force to reform the behavior of the model, which is in the form of user feedback.

2.3.3 IOT Device

Any physical objects that make up the IOT space, except the users, can be considered as IOT devices. The growing technology advancement enables everyday objects to possess more computational capability which supports wired or wireless connections, task execution, and/or computational analysis. This is what triggers the entire field of INTERNET OF THINGS.

To address the functional complexity of the IOT devices, Figure 1 shows a conceptual model that characterizes the different functions of an IOT device aside from the implementation and underlying technology. Next, we discuss each layer more deeply.

The Communication layer encapsulates the standard communication protocol to connect a device to another device. This layer specifically assumes the device implementation of the Open Systems Interconnection (OSI) model (Day and Zimmermann, 1983) through one or more communication technologies. Besides facil-

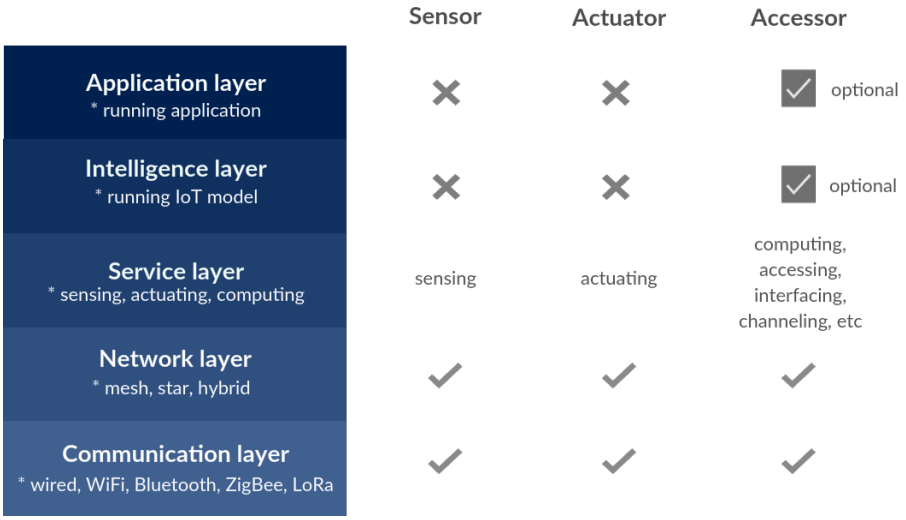

Figure 1: Conceptual Model of IOT device

Table 1: Communication Technologies targeted for the IOT

Technology	Wire	Topology	Speed	Power	Range
WiFi	Wireless	Star	High	High	Moderate
Bluetooth	Wireless	P2P	Moderate	Low	Close
BLE	Wireless	Mesh, Star	Moderate	Low	Moderate
ZigBee	Wireless	Mesh	Low	Moderate	Close
LTE	Wireless	P2P	High	Moderate	Long
LoRa	Wireless	P2P	Low	Low	Long
NFC	Wireless	P2P	Low	Low	Very Close
Thread	Wireless	Mesh	Low	Low	Moderate
Ethernet	Wired	Bus, Star	High	High	Long

itating the communication and the data transfer process, this layer also enables the device's discovery protocols. The proliferation of IOT devices is vast, thus, the efficiency of the Communication layer is essential to define the entire system efficiency.

Motivated by the projected scale of IOT deployment, the exponential growth of communication channels among IOT devices becomes the main drive to make wireless technology more preferred over the wired. Nonetheless, this circumstance does not eliminate entirely wired communication from the implementation. In the industrial level, the throughput requirement is critical and, therefore, the usage of wired connections is more practical. Recently, many new wireless communication

technologies emerge to support the revolutionizing IoT, varying based on operating range, power consumption, and throughput to support different communication needs in the IoT space. Table 1 mentions a handful of wireless technologies targeted for the IoT (Pokhrel and Williamson, 2018, Ensworth and Reynolds, 2017, Ngangue Ndih and Cherkaoui, 2016, Saxena et al., 2016, nfc, 2016, Lan et al., 2019).

The ability to maintain multiple connections in a device is beneficial for reachability and reliability with a cost of higher power consumption. In the implementation, we expect that a device broadcasts only a single unique identity through all of the connections. Therefore, the device is yet recognized as a single entity despite being accessed from different connections. Depending on design decisions and other important considerations, the possibility to deviate from a single-identity requirement is open.

The Network layer defines the network topology, which is the arrangement structure of the IoT devices in a communication network. The information routed from one point to another follows the specified network structure. While the communication layer handles the low-level communication technology between two devices, this layer specifically reckons the structure of device-to-device communications to achieve the essential qualities in an IoT system, i.e., reliability, speed, power, and security.

Referring to Table 1, some communication technologies have had the network topology inherently defined. While each network topology has advantages over the others, WARBLE assumes Mesh Network topology as the prime IoT network topology. This topology offers more integral benefits for this application. We will further discuss the benefits in Section 2.5. However, it does not limit WARBLE to adopt the other topologies to achieve optimization in different aspects.

The Service layer applies a discretization of the device capabilities that specifically contribute to the IoT system. Each capability is also called as a SERVICE. A device's list of SERVICES is announced during device discovery so that all IoT components know its potential contribution. At this layer, a SERVICE may be publicly open or protected to prevent unauthenticated or misuse of the device. Section 2.3.4 explains more details about SERVICE. Most of the SERVICES embrace the commu-

nication layer to perform its functionality. In a sensor, it gathers sensory data and transmits it via a communication channel. In an actuator, it attains the task through the Communication layer. For the upper layers, the Service layer accommodates the device's basic functionalities which are available for further intelligent processes or applications.

The Intelligence layer features one or more IOT models that construct an intelligent IOT space. Through this layer, the IOT model could communicate to other IOT devices to gather data for building context awareness or to execute some actions that enable a responsive IOT space. The IOT model largely uses the SERVICES from the Service layer to perform its functions. This layer is optional as it requires a computational-intensive capability and more power.

The Application layer accommodates the high-level application that uses the SERVICES from the Intelligence layer and below. When the IOT reached its matured form, there would have many application possibilities on how to utilize the sensor data and to manage the actuators, for example, surveillance system or healthcare. Different IOT deployment levels, e.g., industrial and city, have different application-level objectives. The boundary between these applications has to be established to avoid a conflict of interest.

2.3.4 SERVICE

To elevate our understanding of WARBLE's IOT building blocks, we take SERVICE as the basis of IOT system. While most IOT devices have an intrinsic overlap of functionalities, this study theoretically differentiates them into multiple SERVICES with each having a single responsibility. In concept, SERVICE is easily picked up and modular to serve as the building blocks of IOT. Therefore, this concept categorizes WARBLE as a SERVICE-oriented model (Ngu et al., 2017, Yelamarthi et al., 2017). A SERVICE is announced, discoverable, and fulfilled by a physical device. A single device is allowed to deliver multiple SERVICES. By all means, SERVICE is an abstract building block which extends to a heterogeneous collection of concrete SERVICES. To easily comprehend the concept, the following are non-exhaustive main categories of concrete SERVICES.

SENSOR passively works on its surrounding space by detecting and measuring one or more specific space properties. The result is represented in the form of various data types, including numbers, boolean, or enumeration constants. A typical SENSOR has a limited operating scope in terms of area, resolution, range, and others.

ACTUATOR has one or more active influencing capabilities to one or more space properties, for example, an air conditioner influences the temperature, humidity, and matter movement (air movement) by blowing cool air. Although a space property does not influence each other, a physical actuator usually has a design limitation that unintentionally affects multiple space properties at the same time. We can refer back to the earlier example of an air conditioner that influences at least three space properties.

ACCESSOR encompasses all supporting functions to consolidate and manage the IOT system. Its abstraction is more general than a central communication hub of sensors and actuators. When defining the IOT structure, the ACCESSORS are the branches of the IOT structure alongside the sensors and actuators as its leaves. To convey the definition more clearly, the following shows a non-exhaustive list of ACCESSORS.

- **Controller**
  A *Controller* allows a user to naturally interacts with the IOT space. It has a user interface embedded in a physical device. Some examples of the user interface are mobile or desktop apps, control panel, virtual assistant, and many others.

  Via the *Controller*, a user is able to change the IOT configuration, to control the IOT devices with either nearby or remotely-connected connection, and, most importantly, to express the user's explicit intents. These explicit user inputs fill the gap when the IOT model could not make enough analysis from the accessible data. Although the *Controller* is not the only gateway to the IOT space, it denotes a special channel for the user to communicate to IOT devices. In that sense, *Controller* is a complex sensor intrinsically that passively gathers the contexts and intents from its user.

The intent can be delivered to the IOT system as a direct command, a selection in control panel, or a feedback. A personal mobile controller is expected to be the most common form of *Controller*. However, a *Controller* can take many other forms to serve the purpose, such as a pedestrian walk button in a crosswalk (static and public).

- **Central Hub**
  A Central Hub is a single gateway to a group of sensors and actuators. A simple IOT device may have a hard limitation on creating a direct communication to other devices. Any access to the simple device has to be propagated via the central hub. Philips Hue bridge is a concrete example of this type of ACCESSOR.

  Despite WARBLE's standing on a mesh network topology in Section 2.5, a centralized structure may be more appropriate in a certain situation as it provides an encapsulation to a network of devices. Adding a full-fledged communication capability to each sensor and actuator might be expensive and resource-hungry. Therefore, a *Central Hub* allows to control them but it saves the cost.

- **Security Hub**
  Extending *Central Hub*, *Security Hub* creates a security barrier for strategic sensors and actuators. It provides an isolation for security purposes against an unauthenticated access. Figure 4 portrays an IOT structure with a *Security Hub* that virtually has a security domain.

- **Relay**
  In the IOT, the communication among the devices is vital. *Relay* SERVICE provides the functionality to receive data from one end and to pass it to another end without any data alteration. This could extend the range of a network although the radio signal reaches the limit of operating range. This SERVICE is mostly used in a mesh network.

- **Computation Server**
  In a battery-powered device, an intensive computation process is highly undesirable. This ACCESSOR could help by accommodating the generic com-

putational task for the battery-powered device because it is scattered in an IoT space and connected directly to a power supply. The requesting device sends the raw data and the functions, the computation server will return the result.

- **Storage Server**
  Abstracting the IoT devices, users, models, and other components into virtual data requires a database. This Accessor affords the requirement to serve a running database application that stores the abstractions of the IoT components in an IoT space. As an example, a Storage Server for a home automation deployment keeps the abstractions of the home devices, the occupants' preferences, and the occupant interaction patterns. Additionally, the IoT architecture may also affect the quality requirements of the Accessor implementation, e.g., throughput, response time, and number of simultaneous connections.

- **Model Analyzer**
  When a user enters a dark room, it is common to switch on a light to illuminate the room. With this interaction pattern, the general usage of a particular IoT space could be accurately represented as a model in a specific case. This Accessor is designed to drive the IoT model to learn the interaction pattern. Subsequently, with the trained model, it acts as the automated controller of the space.

- **Welcome Server**
  A continuous device discovery is resource-expensive, especially for a mobile device. Placed in a static location, this Accessor continuously discovers the surrounding devices and optimizes a base model to capture the user interaction patterns to utilize the corresponding IoT devices. When a first-time user enters the space, this Accessor delivers a set of comprehensive welcoming data to the user. It also prepares the IoT components for the incoming guest. This mechanism provides the user a smooth IoT experience as if the user had been using the new space for a while. Some concrete data includes IoT general models, IoT system services, and informational guidelines.

In turn, the user feedback on the model can be shared back to the server to improve the base model for future guests.

2.4 Model Architecture

An IOT system can employ multiple models to provide a greater capacity to capture the interaction patterns in order to converge to a close representation of the system. This problem needs an extensive research study to uncover various optimized architectures of the model implementation and to arrive with a structured solution to tackle different situations. Here are some questions that define the architecture:

- What are the components?
- What are the static and the moving parts?
- What is the algorithm of the components?
- Where to put these components?
- What are the input and output of the components?
- How the components are configured?
- What is the general conceptual model to characterize the components?
- What are the data?
- What are the functions?

WARBLE has been exploring different architectures for the IOT model, especially on the model ownership or its coverage area. Figure 2 depicts the architecture distribution based on that criteria. Hypothetically, we can summarize the characteristics of the two extreme ends in the distribution in Table 2.

One architecture projects all IOT models to the user as close as possible, i.e., user's personal mobile device. Thus, the model is autonomous and highly personalized. More user interactions will help the model to perceive the user personal

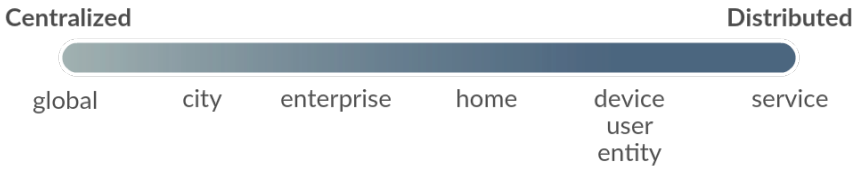

Figure 2: IOT Architectural Paradigms by Coverage Area

Table 2: Centralized vs Distributed IOT Architecture

	Centralized Architecture	**Distributed Architecture**
Implementation	Simpler	Arduous
IOT model complexity	Complex	Simple
Single-point of failure	Yes	Design-dependent
Privacy Management	Centralized	Device-managed
Attack Points	Central Server	All devices
Attack Detection	Solitary	Collaborative

preference. As the user travel within an IOT space, the model constantly discovering new devices and environmental terrain. Subsequently, the model's reactive response generates a series of actions to the IOT system. The WARBLE *middleware* described in Chapter 4 implements this architecture.

Another architecture WARBLE has been studying is a distributed architecture which leverage the IOT model into the appropriate IOT components, i.e., IOT devices and users. This architecture assumes these components as entities which possess their own intents, necessities, and capabilities. All of the entities contribute to establishing the IOT system by working independently and collaboratively as needed. Although it struggles at the beginning, the IOT system would be adaptively progressing to a well-tuned smart system as more interactions and feedback take place.

Personalization is one of the possibilities in the IOT technology. The user story in Section 2.2 can be actualized by abstracting the personalization element into data and functions in the model. A personalized model ensures personalized experiences when the user interacts with the IOT components. Another research in WARBLE explores the possibility to divide a personalized model into a general model and a personalized model delta. There are many benefits adopting this design: the general model is reusable and distributable, the personalized model can

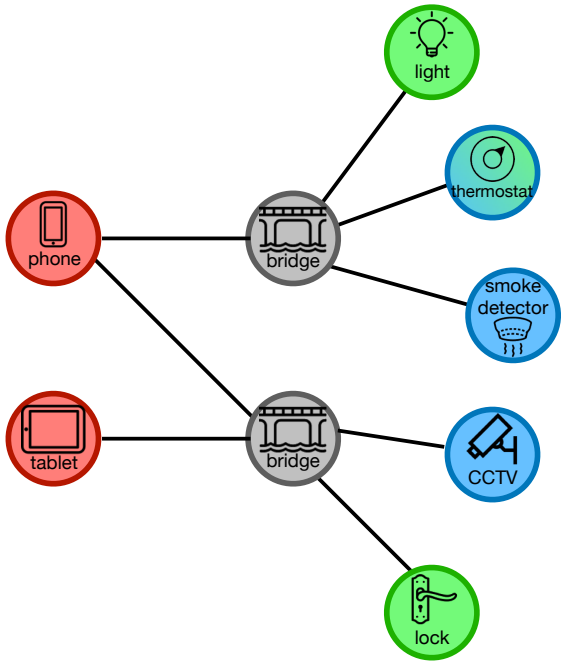

Figure 3: Current Practice of IoT Network Topology

converge quicker, and the privacy is easier to manage.

2.5 Device Network Topology

Most currently available IoT ecosystems adopt a centralized (or star) topology in managing a local group of IoT devices, depicted in Figure 3. The centralized cloud platform can also be utilized to achieve this topology. This topology has several benefits of being simpler, cheaper, and proprietary. Centralized topology is notable by the presence of a central hub in each IoT space level. This structure can also be recursive until the highest central hub. Each central hub connects to a number of sensors and actuators. Therefore, each sensor or actuator is designed to maintain a simple yet secure communication to the central hub. The central hub acts both as the access point and the external gateway for all IoT devices. Despite its simplicity, this topology is rigid and prone to bottleneck situations, prevents interoperability, and has a single point of failure.

On the other hand, WARBLE assumes a mesh network topology that inherently provides more scalability and reliability. However, we also consider other net-

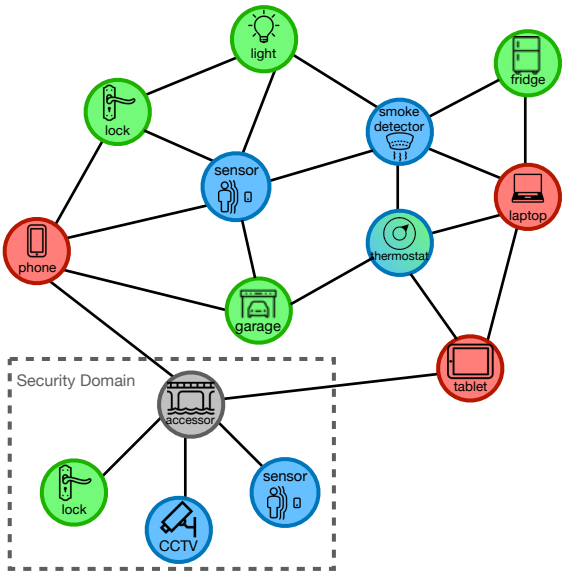

Figure 4: WARBLE-envisioned Mesh Network Topology

work topologies to achieve optimization, such as security, latency, and compatibility. Multiple nearby devices, known as nodes, are web-like connected with short-range communication technologies. Each device is open to connections with other devices and also exposes different SERVICES based on its capabilities to influence the space. Therefore, there may be multiple possible pathways to connect any permutations of two devices, preventing a single point of failure. Mesh network is also capable to self-configure the pathways. This characteristic is favorable when a particular device becomes dysfunctional. Additionally, although the operating range of the radio signal is limited, this topology can still cover a large operating area because each device is able to relay information to other adjacent devices, like building a long chain from multiple short connections. Despite its superiority over the others, mesh network introduces a relatively greater latency due to its *"hopping"* method through multiple devices when routing the data. This is another active research area to find an algorithm that could reduce the latency significantly.

When a local group needs to connect with another remote device, the gateways with Internet capability could provide this SERVICE to the nearby devices. Figure 4 depicts a network that is mainly configured with a mesh topology, but a part of it uses a star topology when security isolation is required for the strategic functions, i.e., functions on the security lock and the CCTV. The ACCESSOR also acts as an

Internet gateway which enables any IOT devices in this network to have access to the Internet. Augmenting more ACCESSORS sparsely to the network increases the network's reliability to connect with the Internet.

As the IOT research progresses, a few emerging wireless technologies embrace this topology paradigm in realizing the IOT, such as Bluetooth Mesh and Thread (refer to Table 1). These technologies then occupy the Network layer and the Communication layer in the IOT device conceptual model referred in Figure 1.

Chapter 3

Related Work

3.1 IoT Middleware

Suffering from the lack of interoperability, the IoT conceives an open problem to solve. There are many other studies that focus on addressing this issue in the IoT today. Web Of Things (Guinard and Trifa, 2009) tackles the interoperability problem by defining a platform-independent API for application developers. The focus is to utilize the current Internet building blocks, like Javascript as the scripting language, HTTP and WebSockets as the data transfer protocol, and JSON as the data encoding format. While this approach resolves the data transferring problem, it assumes the Internet as the sole communication channel. OpenIoT (Soldatos et al., 2014) features an open-source cloud-based middleware which connects sensors to analytic models to actuators. Through OpenIoT IDE, the analytic models can be pre-defined. ELIoT (Sivieri et al., 2016) enables a IoT framework that separates the localized interaction from the internet-wide interactions. It realizes that a cloud-based IoT system, such as Xively (Xively, 2013), ThingSpeak (ThingSpeak, 2010), and OpenSense (OpenSense, 2010), falls short on a stricter performance requirement. A control panel runs ELIoT to manage the interactions of devices in a smart home. The concept of localized and internet-wide interactions is similar to Warble's structure while ELIoT still uses a centralized view locally. Devify (Chen, 2018) complements Warble-envisioned IoT structure to a decentralized IoT architecture to enforce a more secure data flow as well as privacy. However, the study does not include interoperability and personalization.

(Pramukantoro and Anwari, 2018) builds a middleware with a multi-protocol gateway on COAP, MQTT, and WebSocket. Many IoT platforms use these protocols for data transfers among IoT devices. IFTTT (IFTTT, 2011) forms an IoT system with a personalized set of predefined rules. While the simple concept is intuitive, it does not have a natural way to interact with the IoT space and also to support for a higher degree of complexity. Managing a growing number of rules can become overwhelming.

3.2 IoT Framework

CupCarbon (Bounceur, 2016) is a city-scale simulation framework used to design, debug, and validate multiple distributed algorithms to monitor environmental data collection from a wireless sensor network and to create some environmental scenarios, such as fire, gas, mobiles. A variety of communication protocols used in IoT can be simulated by SimpleIoTSimulator™ (SimpleIoTSimulator) and NetSim™. They support MQTT (Jun-Hong et al., 2018), CoAP (Lan et al., 2019), and HTTP with a packet transfer visualization to help with an optimized infrastructure implementation. With the rise of big data from the IoT, the need to process the data increasingly necessary. IOTSim (Zeng et al., 2017) probes into this technology by providing a simulation to build a cloud infrastructure based on MapReduce model (Dean and Ghemawat, 2008). Gazebo (Koenig and Howard, 2004) simulates a 3D space and its properties for robotics simulation. Despite some limitations, it also provide some degree of customization on building an IoT device.

Chapter 4

WARBLE Middleware

This chapter elucidates the WARBLE *middleware*. It is one of the IOT middleware that adopt a user-driven IOT architecture. This architecture encapsulates all active IOT processes, all IOT models, and all algorithms and brings them to the user side. The other IOT devices, including sensors, actuators, and accessors, are passively in standby mode or, at least, have a minimum number of active processes to execute the tasks transmitted by the user. Figure 5 depicts the directed command flow in this architecture. To obtain the explicit inputs and to build a user context, the user carries a portable device that runs the software artifacts. Therefore, the portable device assumes multiple SERVICES as the *Controller* and *Model Analyzer* at the same time.

In the implementation, the WARBLE *middleware* is written in Java and adopts object-oriented programming. The implementation and the testing are done largely in Android platform with Philips Hue and Wink ecosystems. There are a number of terms we use interchangeably throughout this study:

- IOT device - *Thing*

4.1 Conceptual Architecture

We start with Figure 6 that represents the conceptual architecture of a WARBLE *middleware* installed in a mobile device. The middleware is supported by the device's computing capability, communication capabilities, and resources which are able to build the user context (*ContextBuilder*). The *ServiceInterface* helps to provide the interface layer between the WARBLE *middleware* and the resources in the mobile device. App developers create WARBLE applications that can use the WARBLE *middleware* to perform many of its functions, for example, a home automation application via voice commands. The WARBLE applications communicate with the WARBLE *middleware* via the WARBLE *API*. The WARBLE *API* is the facade of the middleware which contains all necessary functions to utilize the middleware features.

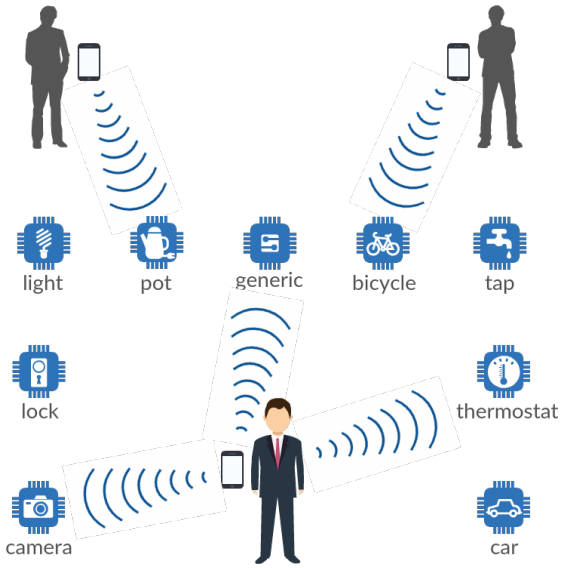

Figure 5: User-Driven IoT Architecture

The WARBLE *middleware* starts with a *Discovery* process. The *Discovery* is proactive, cotinuous, and available, either manually or timer-triggered. Currently, the *Discovery* supports UPnP (Pehkonen and Koivisto, 2010) and SSDP (Wu and Dong, 2006), but extensible to mDNS (Cheshire and Krochmal, 2013), Physical Web (Want, 2015), and HyperCat (hyp, 2015). Since a *Discovery* process may take a considerable amount of time, the WARBLE *middleware* is designed to use multiple threads whenever possible. On the other hand, the *ThingManager* takes a crucial role in the WARBLE *middleware*. It organizes most of the processes that utilize the internal moving parts, including *Discovery*.

The *ThingRegistry* then records the outcome of the *Discovery* in the form of a database. This functionality is an example of *Storage Server* as a SERVICE. The continuous *Discovery* ensures the *ThingRegistry* possesses an updated snapshot of nearby IOT devices. Although some devices are no longer reachable, *ThingRegistry* does not delete the tuples of the unreachable devices. This behavior keeps the prior interaction data and improves the IOT model for a better user experience. After the *Discovery* successfully finds the nearby IOT devices, the WARBLE application may request to interact with them, first by creating a *Binding*.

Binding is a unique WARBLE *middleware* concept that objectifies an interim

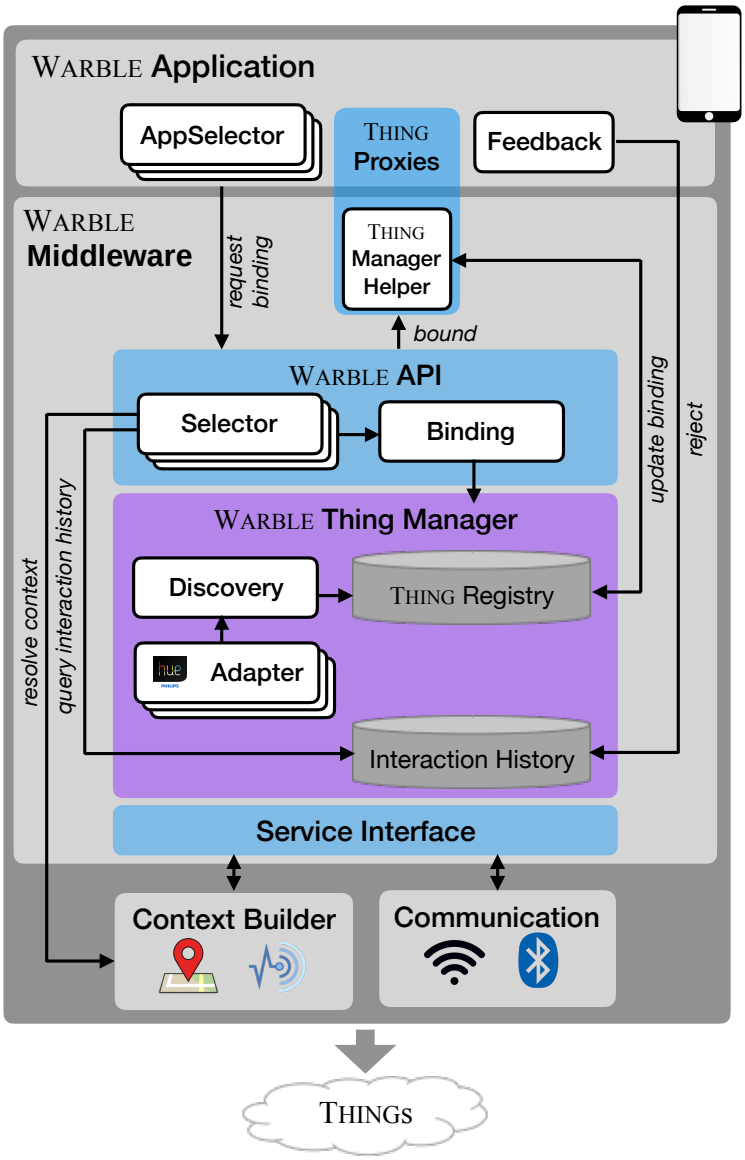

Figure 6: Conceptual Architecture of WARBLE *middleware*

one-to-many connection between a user and the appropriate devices. An example of *Binding* usage assumes a user who needs illumination in a dark bathroom. A *Binding* is virtually created between the user and the two main bathroom lights. The lights are temporarily the most appropriate lights to provide the right amount of illumination as long as the user is in the bathroom. The WARBLE *middleware* materializes the concept of *Binding* in the WARBLE *API* which works together with the *ThingRegistry* to perform its functions.

There are two types of *Binding*, i.e., *OneTimeBinding* and *DynamicBinding*. A *OneTimeBinding* only validates a one-time request to designate the state of multiple bound devices. Afterward, the *OneTimeBinding* is broken. A *OneTimeBinding* is an event-specific *Binding*. In contrast, a *DynamicBinding* request creates a type of *Binding* that allows dynamical user-to-devices connections when there are contextual changes, such as a significant user movement in time and space or a change in user preference. In principle, a *DynamicBinding* is composed into a time series of *OneTimeBinding* which is continuously evaluated.

To clearly distinguish the differences, we could assume the previous example of a user using a bathroom. A *OneTimeBinding* enables the user to switch on the two appropriate bathroom lights. Although the *Binding* "binds" the lights, the WARBLE *middleware* requires the user to explicitly turn on the lights on the WARBLE application. On the contrary, an active *DynamicBinding* maintains illumination wherever the user goes, including entering and leaving the bathroom. The user only specifies the need for illumination explicitly when a *DynamicBinding* is constructed.

The outcome of a *Binding* is *Proxies*. A *Proxy* is a programmatic object representation of a physical device. Because it relates to a particular *Binding*, the set of *Proxies* represents the set of the appropriate physical devices. The WARBLE application could directly interact with the physical devices through their *Proxies*. To support many types of devices and IOT ecosystems, the WARBLE *middleware* implements polymorphism through the *Adapters*. The *Adapter* enables the interoperability in the WARBLE *middleware*. In most IOT ecosystems, each supports only one communication protocol with a unique API and also one discovery protocol.

As WARBLE's assumption embraces mesh network topology with other topolo-

Table 3: Non-exhaustive List of *Commands*

Command	Description
GET_INFO	Return the device basic information, including a universally unique identifier (UUID), SERVICES, and connected devices.
CREATE_CREDENTIAL	Create a new credential (extensible)
AUTHENTICATE	Authenticate with a pre-defined credential (extensible)
GET_STATE	Return the device current state which affects the IOT space, for example, active state, light color, temperature setting.
SET_STATE	Set the device state which affects the IOT space

gies for optimization, the WARBLE *middleware* abstracts the network of the IOT devices to follow this topology. The significant difference to other topologies is the network traversal algorithm. While the *ThingRegistry* holds the entire list of the discovered IOT devices, the *ThingManagerHelper* maintains a subgraph instantiation of the *Proxies*. This allows a performance optimization by accessing the entire *ThingRegistry* every time the WARBLE application requests an action. As the user context changes or the *Discovery* finds new devices, the running *Binding* gets updated as well as the *ThingManagerHelper* and the *ThingRegistry*.

With all these components running inside the WARBLE *middleware*, the WARBLE application may interact to the IOT devices through the *Proxies* by sending a *Command*. Table 3 lists the available *Commands*. Through the *Adapter*, each *Command* is translated to the respective command that is comprehensible by the target device. Subsequently, the communication module transmits the command through the appropriate communication channel.

When we simply turn on an incorrect light in a room, we normally flip the light switch off and try another one. This is a natural interaction for us. The WARBLE *middleware* implements a generalization of this intuitive behavior through *Reject*. The WARBLE application may implement an intuitive *Feedback* interface to provide a *Reject* input to the WARBLE *middleware*. The *Reject* informs the WARBLE *middleware*, especially when the *Binding* is inappropriate. One concrete implementation we foresee is if multiple repeated actions are done in a fairly short amount of time. This behavior implies that the user gives multiple "tries", ex-

pecting the application would give a different set of bound devices for each trial. Ultimately, the user finds the right device.

The transmitted *Commands* indirectly reflect user interactions to the IOT devices. The WARBLE *middleware* stores each of these interactions with the user context in *InteractionHistory*. The *Reject* is also a form of user interaction which is recorded in *InteractionHistory*. Utilizing this valuable information, the WARBLE *middleware* can learn to establish personalization.

By now, we overview the internal mechanism of WARBLE *middleware* and all the moving parts. Next, we are going to describe the essential modules in more details.

4.1.1 The Adapter

Interoperability is one of the common problems in today's IOT technology. Different vendors build their own ecosystem which may potentially introduce scalability or extensibility problem in the future. WARBLE does not confront this situation because this technological infancy which will eventually converge to a better design. Nonetheless, the diversity of the IOT ecosystem, the communication network and protocol, and the high-level programming interface cause a virtual barrier for an application developer to enter the field. Philips Hue API (Philips Hue API, 2017) and Wink API (Wink API, 2017) are good examples of the ecosystem diversity.

Therefore, inspired by Adapter Design Pattern (Beck, 1995), the WARBLE *middleware* utilizes the idea in the application level, i.e., *Adapter*. The *Adapter* is an abstraction layer which transforms the heterogeneity of IOT devices into a programming interface or API in the form of device attributes and functions to the WARBLE application. In this way, the WARBLE applications see assorted IOT devices as a set of uniform devices with different capabilities.

The current implementation of the *Adapter* is done manually by converting each vendor's API to its Java representation. This process is manual by reviewing their API documentation. All API has to be implemented in an *Adapter* for the device to work with the WARBLE *middleware*. Despite its manual process, we could pull up some reusable modules, for example, HTTP request, UPnP discovery protocol (Pehkonen and Koivisto, 2010), OAuth2 authentication framework (Hardt,

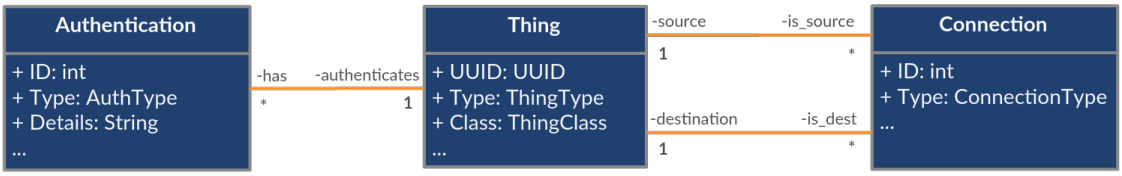

Figure 7: *ThingRegistry* Database Model

2012), and many others.

4.1.2 The Thing Registry

The *ThingManager* is a module which is responsible to manage the programming representation of nearby IoT devices and their connections. Despite WARBLE assumes mesh network topology, most of current IoT ecosystems are not deployed with this topology. Most of them embrace star (centralized) topology. However, the traversal algorithm in the WARBLE *middleware* still works in other types of topology because mesh network is inherently not well-structured, for example, hierarchically or dependent on a single node or connection. Therefore, the algorithm has no issues to traverse in a structured network

The *ThingManager* maintains its knowledge of IoT devices with a graph data structure. The vertices represent the IoT devices whereas the edges represent the device-to-device connections. An edge can be uni-directional or bi-directional. With this requirement, the *ThingRegistry* models the graph into three relational database tables, i.e. *Thing* table, connection table, and authentication table. We refer a *Thing* as an IoT device. Figure 7 describes the database model adopted by the *ThingRegistry* in the WARBLE *middleware*. The *Thing* table records *Thing*'s UUID as the primary key, type, class, etc. The *Thing* Type is a type based on *Thing* functions, like ACCESSOR, Light, Camera, etc. The *Thing* Class refers to a vendor-specific product type, such as VendorB-Light, VendorA-SmokeDetector, etc. The Connection table contains the connections between two *Thing* with a specific communication protocol. The Authentication table maintains credential records of the user to *Things*. Listing 1 exhibits the content of the *ThingRegistry* for an IoT space depicted by Figure 4.

While the *ThingRegistry* keeps the *Things* in database form, the *ThingMan-*

Listing 1: *ThingRegistry* for the IOT structure in Figure 4.
Some fields and values are simplified.

```
Thing Table
UUID  Type             Class                      ...
0001  Light            VendorB-Light
0002  SmokeDetector    VendorA-SmokeDetector
0003  Sensor           VendorC-Sensor
0004  Thermostat       VendorA-Thermostat
0005  Fridge           VendorB-Fridge
0006  Lock             VendorB-Lock
0007  GarageDoor       VendorC-GarageDoor
0008  Accessor         VendorB-Bridge
0009  Lock             VendorB-Lock
0010  Camera           VendorB-CCTV
0011  Sensor           VendorB-Sensor
...
Connection Table
ID  Source  Destination  Type        ...
1   0001    0002         Bluetooth
2   0003    0002         BLE
3   0004    0002         WiFi
4   0005    0002         Bluetooth
...
Authentication Table
ID  Thing  Type          Details              ...
1   0001   UserPassword  myusername, ******
2   0002   Token         **********
```

agerHelper maintains a loaded subgraph of the entire *Thing* network from the same database. The exposed *Proxies* refer to vertices in the subgraph. This strategy is adopted to reduce latency compared to accessing the database directly. Subsequently, the continuous *Discovery* keeps the *ThingRegistry* and the subgraph by the *ThingManagerHelper* to reflect real-time nearby *Things*.

4.1.3 The Selector

As described in Section 4.1, the WARBLE application starts an interaction request with a *Binding*. Then, the *Binding* would internally select the relevant devices from the *ThingRegistry* based on some criteria. The selection is very situational according to the type of requested interaction, for example:

- An interaction to unlock a door most likely selects the nearest door
- An interaction to show the situation at the front door selects the camera that captures the corresponding area

The WARBLE *middleware* utilizes a single fundamental algorithm to select the

relevant devices by executing a database query to the *ThingRegistry*. The query is expressed in one or more *Selectors* written as Java classes. A *Selector* requires zero or more inputs as the predicates from the *ContextBuilder*, such as GPS location, current time, and accelerometer data. In a more profound *Selector*, it may fill the predicates by collecting contextual data from other sensors in the *ThingRegistry* by recursively creating another *OneTimeBinding*.

By default, the WARBLE *middleware* provides several basic *Selectors* to start with. The following is some examples to better understand the *Selector* concept.

- ***AllThingSelector***
  requires no inputs. The output is all reachable *Thing* in the *ThingRegistry*.

- ***TypeSelector***
  requires an input of *Thing* type, such as Camera or Accessor. The output is all reachable *Thing* which matches the *Thing* type.

- ***NearestThingSelector***
  requires an input of user current location. The output is the nearest reachable *Thing* in the *ThingRegistry*.

- ***RangeSelector***
  requires inputs of location and maximum range. The output is all reachable *Thing* within the maximum range measured from the user current location. The location may be a location other than the user current location.

- ***LineOfSightSelector***
  requires an input of location. The output is all reachable *Thing* which senses or actuates the state of the specified location. Some examples include camera, light, and air conditioner.

- ***InteractionHistorySelector***
  requires no inputs. The output is all reachable *Thing* without any records of *Reject* in the *InteractionHistory*.

Since *Selector* is an abstract query concept to the *ThingRegistry*, the WARBLE application could define new *Selectors* by extending the abstract *Selector*.

A *Binding* may handle multiple *Selectors* at the same time. In fact, the basic *Selectors* as listed above are not very useful only in its form. Nonetheless, when the application combines them together, they would serve a more purposeful query. As an example, a *TypeSelector* on Light and a *LineOfSightSelector* on user's location would select the appropriate set of lights to illuminate the area surrounding the user.

4.1.4 The Binding Abstraction

The next step of a WARBLE application's interaction with *Things* is to create and maintain connections to the *Things*, via a process we call *Binding*. As mentioned previously, *Binding* is one of WARBLE *middleware* concepts that objectifies a temporary one-to-many connection between a user and the possible appropriate devices. Conceptually, WARBLE *middleware* presents an abstract operation called `bind`. The WARBLE application provides *Selectors* in a call to a `bind` method, and such a call returns *Proxies* to one or more bound *Things*, assuming a satisfying *Thing* can be identified. This *Binding* abstraction shields the developer from explicitly dealing with the nuts and bolts of connections to *Things*, thereby simplifying programming IOT applications. Conceptually, the abstract `bind` is defined as:

$$\texttt{bind}(\texttt{template}, \texttt{options}, [\texttt{k}])$$

where `template` is a set of zero or more *Selectors*, and `options` dictates the *Binding* logic (e.g., one-time vs. dynamic, additional actions to take on *Binding*, etc.). Because a *Binding* request can select more than one *Thing*, the optional `k` determines the maximum number of *Things* to return. In some cases, there may not exist `k` *Things* that match, so at most `k` items are returned. The default value is one. The result of a *Binding* is an abstract data structure containing *Proxies* to bound *Things*. For simplicity, we treat the data structure as a list, though future work could layer more complex data structures (e.g., placing *Things* on a map of physical space).

OneTimeBinding Requesting a *OneTimeBinding* requires no *Binding* options in `bind`. We refer to the concrete action as `fetch`, since it simply fetches *Proxies*

for *Things* that match the `template`. Upon binding, the *ThingManager* builds a subgraph containing the currently available ACCESSOR paths from the *Controller* making the request to each selected *Thing*. This subgraph is given to the *Proxy*'s *ThingManagerHelper*, which, when the application interacts with the *Thing* via the *Proxy*, finds a path to the *Thing* in the graph. The *ThingManagerHelper* also manages any necessary authentication via the ACCESSORS in the graph. If the paths to the *Thing* bound to a *OneTimeBinding* are no longer traversable when an application attempts to interact with the *Proxy*, WARBLE *middleware* returns an exception. It is entirely in the hands of the application (and user) to ensure that the *Thing* is placed in a consistent end state if necessary (e.g., turning a light off when the application is finished with it) before the *Thing* becomes unreachable.

Some applications may desire to immediately execute a fixed sequence of operations on a set of selected *Things*. In WARBLE *middleware*, `batch` allows the application to provide such a sequence of operations, removing the need for the application programmer to manage *Thing Proxies*. When invoking the `batch` *OneTimeBinding*, the application provides an `onBind` parameter, which is effectively a script of the actions to take upon *Binding* to each selected *Thing*. Instead of returning a *Proxy* to the application, a `batch` operation binds the selected *Things* then executes the actions directly, internally handling exceptions (such as disconnections). When the `onBind` actions have completed, the bindings effectively go out of scope.

DynamicBinding In a *DynamicBinding*, as WARBLE *middleware*'s *Discovery* mechanisms update the *ThingRegistry*, the specific *Things* behind a *Proxy* may be updated. This is handled in each *DynamicBinding*'s *ThingManagerHelper*, which listens for and responds to updates in the *ThingRegistry*. For instance, if the application is connected to the two *closest* light *Things*, when new lights are discovered, WARBLE *middleware* notifies the *ThingManagerHelper*, which checks whether the new lights are closer and updates the *Proxy* bindings as needed. If the connectivity paths to a bound *Thing* change, the graph embedded in the *Proxy*'s *ThingManagerHelper* is similarly updated.

A *DynamicBinding Proxy* is *read-only*. To *change* the state of a *DynamicBinding Proxy*, applications specify a `plan` that defines the desired continuous state

of bound ACTUATOR *Things*. When new *Things* are bound to the *Proxies*, the *ThingManagerHelper* automatically adjusts their state using the instructions encoded in the `plan` without explicit application interaction. Conceptually, this `plan` is provided within the *Binding* `options` of the *DynamicBinding*'s `bind` operation; practically, the WARBLE *middleware* API allows specifying the `plan` separately from the *Binding*, which also allows the `plan` to be updated without tearing down a *DynamicBinding* and building a new one. When WARBLE *middleware* updates a *DynamicBinding*, prior bound *Things* are *unbound*; upon unbinding the *ThingManagerHelper* returns unbound *Things* to their state before they were bound to the *DynamicBinding*.

4.1.5 The *InteractionHistory*

WARBLE *middleware*'s *InteractionHistory* stores information about interactions with particular *Things* in particular contexts. Each user device maintains its own *InteractionHistory*. This is an intentional design decision to personalize the use of prior interactions per user; placing this information in a shared repository may violate privacy. Further, what constitutes a successful interaction for two users may be different. Associating the *InteractionHistory* with a user is therefore motivated by our goal of personalizing the IOT. On the other hand, there are good reasons for sharing *InteractionHistory* among users who interact with the same *Things*. Sharing may lead to faster learning, especially in spaces that are new to a user or with which the user interacts only rarely. Therefore, an alternative design could push (some of) the *InteractionHistory* to *Things* themselves, leading to the emergence of an IOT infrastructure that learns about itself and its ability to support applications.

The *InteractionHistory* maintains entries for all actions performed by the application on a binding, including an entry each the time a dynamic binding's `plan` changes the state of some bound thing. Each entry represents a successful or unsuccessful action taken on a *Thing* that was bound based on some `templates`. We associate a timestamp to each piece of *InteractionHistory*, and applications can tailor *Selector* behavior based on these timestamps.

Entries in the *InteractionHistory* are initialized during selection, but because entries require information collected as the application interacts with a bound *Thing*, they must be updated over time. The complete process is:

1. An application initiates a `bind`, including the `template`. This creates a pending *InteractionHistory* entry (noted with the *p* subscript) for each *Thing* that matches the selection:

$$(id, \texttt{template}, Thing)_p$$

   id is a unique identifier for the *InteractionHistory* entry.

2. The application interacts with the *Proxy*, either directly, in the case of a *OneTimeBinding* or indirectly through the `plan`, in the case of the dynamic binding. The *ThingManagerHelper* captures each interaction (e.g., each method called on the *Proxy*), copies the related pending entry, marks the copy as complete (noted by the c subscript), and notes the specific `action` taken and its timestamp. The `action` includes the context of the interaction; in our current implementation, we simply log the *Controller* device's location. Multiple interactions create multiple completed entries in the history. WARBLE *middleware* assumes that the interaction is successful until notified otherwise:

$$(id, \text{time}, \texttt{template}, Thing, \texttt{action}, true)_c$$

3. An application may give explicit feedback by invoking *Reject* on a *Thing* for its most recent `action`, marking the entry unsuccessful (i.e., changing *true* to *false*):

$$(id, \text{time}, \texttt{template}, Thing, \texttt{action}, false)_c$$

A *Reject* notice also initiates a rollback, which examines the rejected action (e.g., `camera.turn(45)`) and performs the logical undo-action (e.g., `camera.turn(-45)`). These undo actions are specified in the WARBLE *middleware Thing Adapter*.

It is now possible to see how WARBLE *middleware* can use the *InteractionHistory* to influence selection. For example, when invoked given a reference location and time range ($[\texttt{t}_{\texttt{start}}, \texttt{t}_{\texttt{end}}]$), a WARBLE *middleware Selector* can examine each *Thing* in the registry that satisfies the applications' `template`. For each matching *Thing*, the selection retrieves relevant *InteractionHistory* tuples (i.e., those about the selected *Thing* whose `timestamps` are within the range) and examines whether interacting with the *Thing* is expected to be successful at the given location.

Table 4: Warble API methods summary

Method	Description
Instantiation	
Warble()	creates a WARBLE *middleware* instance to access its API; initiates continuous discovery in the background
Discovery	
warble.discover()	performs a full discovery process manually
OneTimeBinding	
warble.fetch(template, [k])	fetches `k` *Things* matching requirements in `template`; provides the abstract `bind` for a *OneTimeBinding*
warble.batch(template, onbind, [k])	selects `k` *Things* matching requirements in `template` and executes the actions listed in `onBind` immediately
warble.reject(*Things*)	indicates a mismatch in *Binding* for each of the provided *Things*
DynamicBinding	
warble.dynamicBind(template)	creates a *DynamicBinding* instance for *Things* matching the requirement in `template`
dBind.fetch([k])	fetches `k` *Things* matching requirement in `template` for *DynamicBinding*
dBind.bind(plan)	starts the *DynamicBinding* to serve the configuration in `plan`
dBind.unbind()	stops the *DynamicBinding*
dBind.reject([*Things*])	indicates a mismatch in *DynamicBinding* and triggers another *Binding* process
Plan	
Plan()	creates a Plan instance
plan.set(preferenceKey, value)	sets value of key identified by preferenceKey (e.g. `Plan.Key.LIGHTING_ON`)
plan.unset(preferenceKey)	unsets a preferenceKey
plan.unsetAll()	clears all preferenceKey settings

4.2 Implementation: Programming with WARBLE *middleware*

We have implemented WARBLE *middleware* on Android (in Java); its API is in Table 4. WARBLE *middleware* and examples available at `https://github.com/UT-MPC/Warble3`. We next walk through several use cases relating to smart lights with the specific aim of demonstrating how WARBLE *middleware* achieves the design goals of interoperability and personalization. These examples also demonstrate how developers use WARBLE *middleware* to simplify the programming task; the next section quantifies these details. We use lighting examples because they are straightforward and accessible; additional examples with other types of SENSORS and ACTUATORS are available. All told, WARBLE *middleware* currently supports 11 types of *Things*. We have integrated four concrete *Things* from three vendors with four different discovery mechanisms.

Personalization in WARBLE *middleware* Unlike a centralized view of the IOT in which one program controlled by the space determines *Things'* behavior, WARBLE *middleware* provides abstractions to allow individual applications to use *Things*

directly. A WARBLE application interacts with the *ThingRegistry* to determine *at runtime*, what the best *Things* are for a given need. In contrast, existing IOT middleware create program scripts *at compile time* that choreograph the behavior of the space's *Things*. This is a subtle shift; WARBLE *middleware* is appropriate for application interactions that are highly adaptive and context dependent. In contrast, existing approaches like Node-RED (Node-RED, 2013) and Calvin (Persson and Angelsmark, 2015) are more suitable for these predictable and scriptable interactions. WARBLE *middleware* is more similar to middleware that take a service-oriented approach (Soldatos et al., 2015). These approaches provide web-programming based interfaces to *Things* in which applications construct and invoke web requests. In contrast, WARBLE *middleware*'s *Binding* and feedback concepts provide a higher-level of programming abstraction more tailored to interacting with *Things* in an object-oriented way. Further, the service-oriented IOT middleware are designed to run on heavyweight cloud infrastructure and are not compatible with an approach that executes entirely on lightweight edge devices (Ngu et al., 2017).

Interoperability in WARBLE *middleware* Once *Things* are available within the WARBLE *middleware* *ThingRegistry*, all WARBLE applications interact with all things using the same set of interfaces (as shown in Listing 1). This approach is very similar to the many other existing efforts at semantic mapping of *Things* to software interfaces (Eisenhauer et al., 2009, Wang et al., 2010, Tan et al., 2010, Yap et al., 2008). Our current implementation of the Service Interface at the bottom of Figure 6 is, therefore, a straightforward mapping of vendor classes to WARBLE *middleware* types. However, it is a straightforward extension of the Service Interface to also enable mapping from other semantic descriptions (e.g., SensorML (SensorML, 2007) or OWL (OWL, 2012)).

Concrete Things and Selectors Listing 1 shows several examples of adapters, which are provided to WARBLE *middleware*'s *TypeSelector* when creating a `template`. WARBLE *middleware* also currently has three context selectors. The *NearestThingSelector* takes a `location` and selects the *Things* closest to the `location`. The *RangeSelector* requires returned *Things* to be within the specified `range` of the

Listing 2: Application-defined selector based on line of sight

```
1  public class LOSSelector extends AbstractSelector {
2    public LOSSelector(Location location, double heading,
3                       double angle, double range) {
4      // ... save instance variables
5    }
6    @Override
7    public List<Thing> select() {
8      CircleSector sector = // ... compute sector
9      List<Thing> reg = Warble.getInstance().getThings();
10     List<Thing> selectedThings = new ArrayList<>();
11     for(Thing thing : reg)
12       if (sector.contains(thing.getLocation())
13         selectedThings.add(thing);
14     return selectedThings;
15   }
16 }
```

provided `location`. The *InteractionHistorySelector* returns *Things* for which the *InteractionHistory* has not logged a negative interaction at the given `location`. Below, we also develop a line of sight selector as an example of how applications can add new *Selectors* to WARBLE *middleware*.

Adding New Things to WARBLE *middleware* The `Thing` base class assumes every *Thing* has a UUID, location, a set of discovery mechanisms, and a set of (direct) connections to other *Things*. WARBLE *middleware* employs the Command design pattern (Beck, 1995), so every *Thing* must also implement a `callCommand` method that handles requests to change the state of the *Thing* (for ACTUATORS) or to retrieve the state of the *Thing* (for SENSORS). Integrating a new *Thing* simply requires selecting the appropriate *Adapter* (e.g., `Light` or `SmokeDetector`) and overriding the `callCommand` method. This override is simplified by `State` definitions for each of *Adapter* type (e.g., `LightState`, etc.) and a `setState` method in the `Thing` base class.

Adding new *Selectors* WARBLE *middleware*'s *Selectors* can be extended to implement application-specific selection. An example is a line of sight *Selector*, whose code is in Listing 2. The author of the *LineOfSightSelector* (1) extends WARBLE *middleware*'s *AbstractSelector*; (2) defines a constructor, which takes parameters required to scope the selector; and (3) overrides the *AbstractSelector*'s `select` method, which encodes the selector logic, in this case selecting *Things* within the sector of a circle centered at `location`, with a radius of `range`, given the `heading` and

Listing 3: Application code for a WARBLE *middleware OneTimeBinding*

```
1  Warble warble = new Warble(); //initiates discovery

2  List<Selector> template = new ArrayList<Selector>();
3  template.add(new TypeSelector(THING_CONCRETE_TYPE.LIGHT,
4                                THING_CONCRETE_TYPE.THERMOSTAT));
5  template.add(new NearestThingSelector(myLoc));
6  template.add(new InteractionHistorySelector(myLoc));

7  List<Thing> things = warble.fetch(template, 3); //things contains the Proxies

8  for (Thing thing : things) {
9    if (thing instance of Light)
10     ((Light) thing).on(); //simplified, Command Pattern
11   else if (thing instance of Thermostat)
12     ((Thermostat) thing).setTemperature(298);          //in Kelvin
13 }
```

angle. An application can use this *Selector* when creating a `template`; the selection process combines it with the rest of the *Selectors* in the `template` to determine which *Things* to engage in the *Binding*.

Programming with WARBLE *middleware* Binding is driven by application-supplied `templates` that state the requirements of selection. Listing 3 shows an application initiating a *OneTimeBinding* to select the three closest `Light` or `Thermostat` *Things* for which the *InteractionHistory* has no record of *Reject* actions at the location `myLoc`. After constructing the `template`, the application invokes `fetch`, which returns a `List` of *Proxies*. The application interacts with these *Proxies* using the `Light` and `Thermostat` *Adapter* interfaces. Listing 4 shows a (partially elided) native implementation of the same functionality, albeit without the *InteractionHistory*. The application itself must directly call discovery and directly implement the detailed selector logic. This approach is unwieldy for the developer, error-prone, not future-proof, and does not provide generic forms for interacting with *Things*.

Listing 5 shows a WARBLE *middleware DynamicBinding* using the same `template`. The application's `plan` sets a `Light` to on. This `plan` is executed any time the *Proxy* is bound to a `Light`. The `plan` also sets the ambient temperature; this is executed any time the *Proxy* is bound to a `Thermostat`. The use of the `plan` for specific *Thing* types is implicit; a `Thermostat` simply does not understand the command `LIGHTING_ON`, so it is ignored. Listing 5 also shows the application updating the `plan`, which updates the state of bound *Things* and changes the *Binding* behavior for future bound *Things*.

Listing 4: (Partial) Native implementation of *OneTimeBinding*

```
1  public void useThings() {
2    int [] types = {THING_CONCRETE_TYPE.LIGHT,
3                    THING_CONCRETE_TYPE.THERMOSTAT};
4    // ... other variables, e.g., location, threshold, etc.

5    // rely on underlying discovery mechanism
6    List<Thing> discoveredThings = Thing.discover();
7    List<Thing> selectedThings = selectType(discoveredThings, types);
8    selectedThings = selectLocation(selectedThings);

9    for (Thing thing : selectedThings)
10     // ... same as lines 9-13 in Listing 3

15 }

16 List<Thing> selectType(List<Thing> things, int[] types){
17   List<Thing> selectedThings = new ArrayList<>();
18   for (Thing thing : things)
19     if (types.contains(thing.getThingConcreteType()))
20       selectedThings.add(thing);
21   return selectedThings;
22 }

23 // return things within a threshold distance
24 List<Thing> selectLocation(List<Thing> things){/*...*/}
```

WARBLE *middleware*'s programming simplification is even starker for the *DynamicBinding*. Listing 6 shows natively implemented code that is similar in functionality to the WARBLE *middleware DynamicBinding* in Listing 5. As shown the native application must implement a form of runnable thread that periodically invokes discovery of *Things* and manually enacts the application's desired behavior (i.e, the WARBLE *middleware* `plan`) on the discovered *Things*. Omitted from this listing is all the necessary exception handling code and how the application handles *Things* it was using that have gone out of scope. All of these behaviors are handled automatically in WARBLE *middleware*; none of the *DynamicBinding* code for the WARBLE *middleware* interaction is elided in Listing 5.

InteractionHistory Conceptually, we view the *InteractionHistory* as a single unit as shown in Figure 6. The implementation, however, contains two distinct components: the *local* and *global* histories. Each *Controller* maintains its own local history in main memory. The local history functions like a cache and 1) provides fast access to entries that will be duplicated (e.g., pending entries copied when an application interacts with a *Thing*) or updated (e.g., completed entries updated upon a *Reject*) and 2) maintains `action` objects that contain code for rolling back an `action` in case

Listing 5: Application code for a WARBLE *middleware DynamicBinding*

```
1  Warble warble = new Warble(); //initiates discovery

2  List<Selector> template = new ArrayList<>();
3  template.add(new ThingConcreteTypeSelector( THING_CONCRETE_TYPE.LIGHT,
       THING_CONCRETE_TYPE.THERMOSTAT));
4  template.add(new NearestThingSelector(myLoc));
5  template.add(new InteractionHistorySelector(myLoc));

6  Plan plan = new Plan();
7  plan.set(Plan.Key.LIGHTING_ON, true);
8  plan.set(Plan.Key.AMBIENT_TEMPERATURE, 298);

9  DBinding dBind = warble.dynamicBind(template, 3);
10 dBind.bind(plan);  // start binding based on plan
11 // ...
12 plan.set(Plan.LIGHTING_COLOR, "RGB#EEEEEE");
13 dBind.bind(plan);  // bind again with changed light color
```

Listing 6: (Partial) native implementation of dynamic binding

```
1  public class ContinuousThread implements Runnable {
2    public ContinuousThread(/* ... */){/* initialization */}
3    public void run() {
4      List<Thing> discoveredThings = Thing.discover();
5      List<Thing> selectedThings = //...Listing 4 Lines 8-9

7      // manually enact the plan
8      for (Thing thing : selectedThings)
9        // .. same as lines 9-13 in Listing 3

14     // continuously recheck the surrounding things
15     while (!stopFlag) {
16       discoveredThings = Thing.discover();
17       newSelectedThings = // select again by type/location
18       if (!newSelectedThings.equals(selectedThings)){
19         // manually enact the plan on any new things
20         // put old things in their original states
21       }
22       selectedThings = newSelectedThings;
23       // exception handling omitted
24       Thread.sleep(rediscoveryPeriod);
25     }
26   }
27 }
```

of a *Reject*.

In contrast, the global history resides on-disk and is accessible to and maintains histories for all WARBLE *middleware Controllers* on the user device. Entries that are unlikely to be updated or duplicated are flushed from the local history to the global history. Code (such as a rollback operation) is not stored in the global history; rollback should not be required on entries in the global history. When to flush entries is an open research question; for now, WARBLE *middleware* flushes entries prior to new selection operations because selection may rely on the availability of the information in the global *InteractionHistory*.

Chapter 5

MESH

After simplifying the complex design problem of INTERNET OF THINGS in Chapter 2, WARBLE extends its reach to design the IOT architecture and the IOT model. WARBLE has been studying a number of different approaches to architect a solution for building the IOT-enabled space. Ideally, the outcome of the research has its evaluation performed in a real IOT space, engaging real devices and real user interactions. However, there may be many implementation issues arise during the setup, especially the time-related and the cost-related issues. Off-the-shelf IOT devices may not have the desired flexibility for the research and, therefore, a complete self-prototype is paramount to continue with the experiment. The barrier of learning new side technical knowledge might also become a hurdle. Instrumentation of an IOT space requires several thorough assessments to provide the right data extraction process. Additionally, the need to perform an extensive preparation for real human interaction experiment is also essential to extract useful dataset.

MESH targets to alleviate these research and development issues. It provides an extensible framework to model an open IOT space which is ready for a quick instrumentation. The objective is to enable faster and easier architecture modeling while being as close as possible with the real environment. By design, the comprehensiveness in MESH is developed with the WARBLE vision in mind. The following is the list of user stories of MESH.

- To render a closed system with a 3-dimensional space and its IOT components
- To prototype an IOT device programmatically
- To emulate human-to-device and device-to-device interactions with time
- To design an IOT architecture to manage the simulated IOT components
- To engineer and execute IOT models on any IOT devices

5.1 MESH Use Case

This section describes a use case that highlights the purpose of MESH to be more tangible.

A researcher considers an idea of IOT *architecture only for a large building. It discretizes the entire closed space into multiple chambers. A chamber is defined as a partially-hollow space bounded by walls. Every chamber has an* ACCESSOR *with a* Model Analyzer SERVICE *to govern the interactions within the chamber. The* ACCESSOR *has the final word to decide what the best action for the user is. But it needs inputs from the other* IOT *devices. Every* IOT *device in the chamber also runs its own model internally and gives an independent recommendation to the* Model Analyzer *for decision making. The researcher has a dataset of prior user interactions with those* IOT *devices.*

To see how this architecture idea performs in the chamber level before proceeding to the building level, the researcher can implement the architecture in MESH. *He creates a* MESH *system with a certain dimension that represents a chamber. Then, he creates a mock* ACCESSOR *with a* Model Analyzer SERVICE *and other devices in the chamber. He also needs to programmatically implement the right model into the devices. Each device is either* ACCESSOR, SENSOR, *or* ACTUATOR *with the capability to run its recommendation model in real time. By replaying the user interactions, the fresh model in the* Model Analyzer *may learn and practice these prior interactions so that it could predict the appropriate actions when a similar context is detected. Depending on availability, some devices might have been provided by default in* MESH. *However, the rest must extend the super device. Then, the* IOT *model can be programmed into the devices.*

During the simulation, the SENSOR *can get sensory data from the simulated space properties while the* ACTUATOR *can manipulate the simulated space properties. According to his architecture, all devices are running their own recommendation models. The* ACCESSOR *gathers the recommendation inputs by utilizing* MESH *communication channels. Then, its model makes an informed decision and executes the generated actions on behalf of the user.*

The researcher can issue a time manipulation command to run step by step or continuously. This feature allows the researcher to probe many attributes in the system, especially at the critical point of time. In result, he analyzes the performance of the under-test IOT *architecture and the* IOT *model. The processes can be iterated until the researcher are satisfied with his architecture.*

5.2 Conceptual Architecture

Figure 8 depicts the conceptual architecture of MESH application. To the external, MESH provides a running command-line interface, the MESH *API*, to interact with all MESH features. The visualization is not the primary target in MESH, but the MESH *API* could generate the input data for the visualization application, especially to visualize the IOT system. After the user starts MESH, the MESH *API* would run in an indefinite loop that waits for an incoming command. The MESH *API* module handles all incoming commands to and outgoing responses from the MESH *Core*.

A MESH user usually starts by constructing a *System*. There are two ways to construct: (1) using MESH *API* commands to construct a *System* from scratch, adding the components one by one, (2) loading from an input file in JSON format (JSON, 1999). For both ways, the *SpaceFactors* assumes a role to handle the *System* construction. The *System* consists of a *Space*, multiple *Entities*, and multiple *Channels*. The *Space* not only defines the *System* spatial dimension but also contains the *SpaceFactors*, such as matter composition and temperature. Next, we add the *Entities*, i.e., IOT devices and also the IOT user, into the *System*. The sensors get contextual data from the *SpaceFactors*, while the actuator imposes a change to the *SpaceFactors*. Lastly, we define the *Channels* to connect device-to-device and device-to-user. That may include discovery processes, data transfer processes, or power transfer processes.

After the *System* is ready, the *SpaceFactors* handles the API commands to probe any attributes in the *System*. This is useful to provide the MESH user or a visualization application with evident data of the *System*. On the other hand, the *SpaceFactors* could manipulate the state of the *System*. Until this point, MESH is ready to execute the active interactions in the *System*. We could start to evaluate our design, IOT models, IOT device prototype, etc.

In the simulation, we manipulate the time by using the time manipulation commands, like `runstep/rs`, `run`, and `stop`. The *Time Keeper* module runs the simulation by *timestep*, where a *timestep* is a preset time interval for running the simulated interactions. In the case that a `runstep/rs` command is used, the *Time Keeper* will run the simulation for one *timestep*, then stop. However, a `run` command will run continuously a *timestep* over another until `stop` is received. By default,

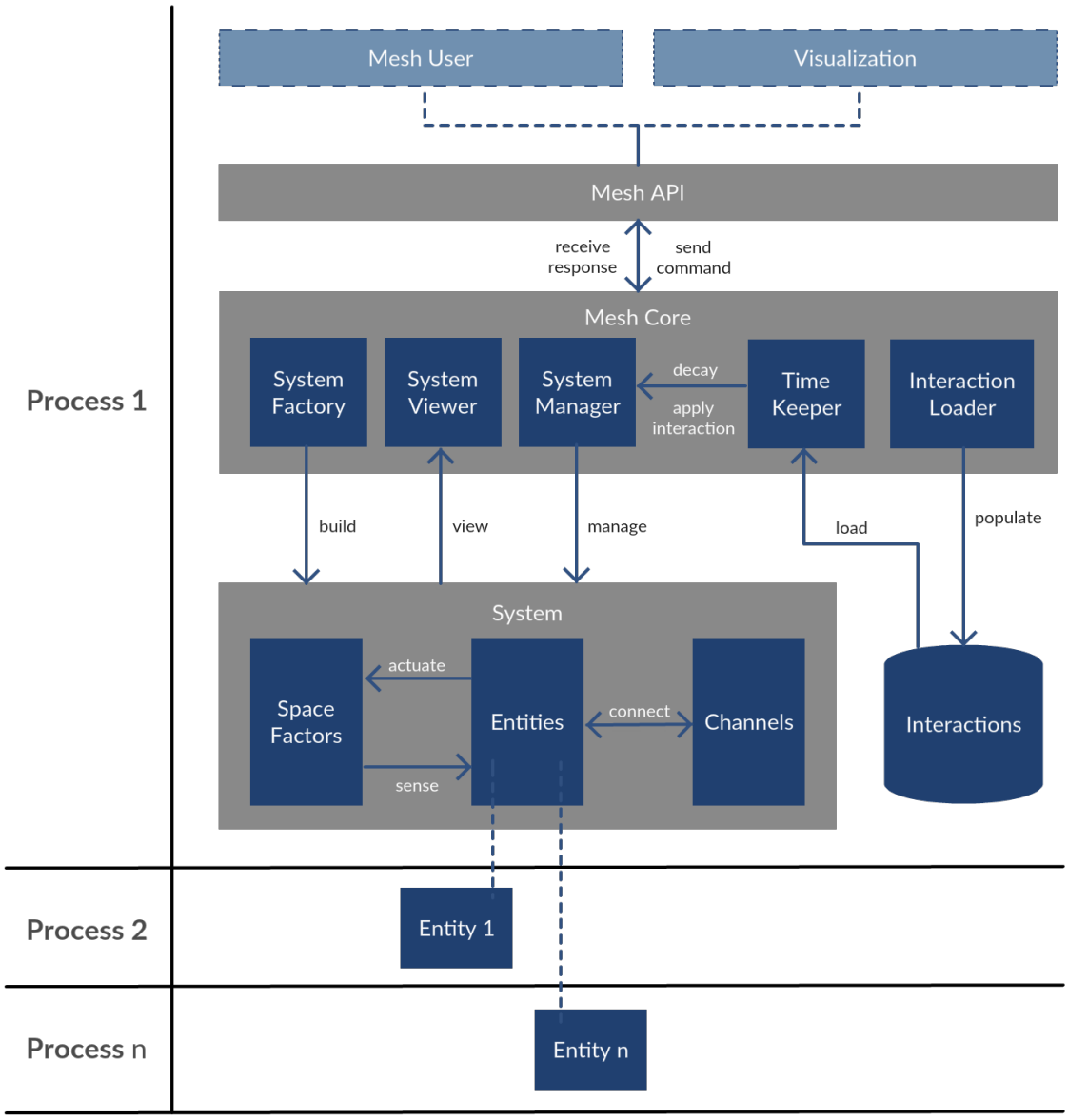

Figure 8: MESH Conceptual Architecture

one *timestep* is one second but it is modifiable at runtime. Algorithm 1 shows processes that are executed for a single *timestep*.

After internal initializing processes, *Time Keeper* runs all *Actuate Functions* and all *Sense Functions* of all *Entities* in order, if any. These *Functions* will be described further in Section 5.3.3. Subsequently, the *Time Keeper* waits indefinitely for any manual interactions from the MESH *API* until the user issues another `run-step/rs` command. The *Time Keeper* skips this step if a `run` command is used. Next, all loaded interactions whose timestamp is within the *timestep* are executed.

Algorithm 1: Algorithm of a Single Iteration in a *timestep*

```
Result: void
initialization;
for entity in entities do
    entity.actuate() // run Actuate function
end
for entity in entities do
    entity.sense() // run Sense function
end
wait for manual interactions from MESH API (skip in "run" command);
for interaction in inTimeInteractions do
    execute interaction
end
for spacefactor in spacefactors do
    decay spacefactor
end
termination;
```

Afterward, each *SpaceFactor* runs a decay function before continuing to the next step. A decay function is a function which follows the physics laws on a *SpaceFactor* to achieve the equilibrium with time. As an example, the positive *Luminosity* in an IOT space would disappear instantly when the light source is switched off. In another example, temperature differences in multiple locations will affect *MatterMovement* to disperse the heat evenly. The final step is to run the internal termination functions before moving to the next iteration.

All of these sequential steps are executed in the main computer process. However, MESH can delegate the internal computation in the *Entities* to other computer

processes as if they are in the real deployment. This is fulfilled by *Compute Function*, refer to Section 5.3.3 for more details. The main target of this feature is to run the device's IOT model in a separate process.

5.3 MESH Main Features

When we need a higher degree of luminosity, we turn on the lights. When we suffer from hot temperature, we turn on the air conditioner. Inspired by how our reasoning turns contextual data into actions, we recognize that there is a necessity to manipulate one or more space properties (also known as space factors), such as luminosity and temperature. Then, we disjoint the complex system into multiple encapsulated components or layers which cohere to our reasoning when doing the human-device interaction. The current MESH primarily focuses on the basic environmental necessities like the *SpaceFactors*. However, the non-environmental necessities, for instance, the feeling of thirst, the will to listen to music, and the demand to go to another place, have separate abstract property layers that are non-spatial. While most of our IOT focus is related to space properties, the idea may be extensible to non-space properties. This may be an unresolved research topic. Furthermore, the idea, which disjoints the complexity in this way, can also be applied to building the IOT architecture and the IOT models.

5.3.1 The *Space* and The *SpaceFactor*

As mentioned previously, a SENSOR or an ACTUATOR performs a function on one or more space factors in the *Space*. Therefore, MESH encapsulates each into a *SpaceFactor* so that we can have a better view of a single space factor alone. A *SpaceFactor* is composed of one or more *SpaceSubfactors* which are highly dependent on each other. In the case that a *SpaceFactor* can be represented with only one *SpaceSubfactor*, the *SpaceFactor* is technically equal to the *SpaceSubfactor*. Otherwise, while we can perform an analysis on the *SpaceSubfactor* level, a *SpaceSubfactor* is not supposed to stand alone in representing the space property.

A 3-dimensional array represents a *SpaceSubfactor*, with the exact dimension of the *Space*. Each element of the array represents a space sub-property of a single

unit cube in the IOT space. The element can take a value of Integer, Float, String, or Boolean. The current structure of the *SpaceFactors* is summarized below:

- *MatterComposition*
  - MatterType
- *Luminosity*
  - Hue
  - Saturation
  - Brightness
- *Temperature*
  - Temperature
- *Humidity*
  - Humidity
- *MatterMovement*
  - X axis
  - Y axis
  - Z axis

The following is the list of *SpaceFactors* which are currently supported in MESH.

MatterComposition represents the presence of matter in the *Space*. As light, heat, radio waves, electricity propagate differently through different matters, this *SpaceFactor* is essential to be the base of the other *SpaceFactors*. The matter types are classified into four different categories listed in Table 5. At the category level, we can have Ether to be displaced by the others, Gas to be displaced by Liquid and Solid, and Liquid to be displaced by Solid only.

Table 5: MESH Matter Types

Category	Matter Type	Description
Ether	Ether	a non-existent hypothetical matter OR vacuum
Gas	Atmosphere	breathable air surrounding us transparent
	Smoke	gas with combustion residue, opaque
Liquid	Water	transparent
Solid	Organic	a collection of organic matter composing living organisms non-heat conductive, non-electrical conductive, opaque
	Plastic	non-heat conductive, non-electrical conductive, opaque
	Wood	non-heat conductive, non-electrical conductive, opaque
	Glass	heat conductive, non-electrical conductive, transparent
	Concrete	non-heat conductive, non-electrical conductive, opaque
	Metal	heat conductive, electrical conductive, opaque
	Perfect Solid	impenetrable

Luminosity The *Luminosity SpaceFactor* is useful for evaluating the interactions pertaining to visible light. It comprises of three *SpaceSubfactors*: Hue, Saturation, and Brightness. All of them takes an integer as the value of the element. Hue ranges from $0 - 360$ whereas Saturation and Brightness range from $0 - 100$. This *SpaceFactor* interacts closely with the *MatterComposition* because the presence of a matter might change the light propagation, such as absorption, reflection, or refraction. The *Luminosity SpaceFactor* also decays with time in an instant. The external environment might change this *SpaceFactor* also, for example, the sun ray enters from a glass window.

Temperature The amount of heat in a discrete space decides the temperature. In MESH, it is represented by an integer value in Kelvin. Most of the IOT devices generate heat when active. Although they are sometimes negligible, we should take the effect into account. Some examples of IOT device affecting *Temperature SpaceFactor* are an air conditioner, stove, oven, light, and personal computer. The decay function of temperature (heat) depends on the external environment. If the *Space* does not allow the heat to escape, for example, due to a solid wall, the *Temperature SpaceFactor* does not decay.

Humidity The *Humidity SpaceFactor* refers to the relative humidity of the air, expressed in percentage. Air conditioner and dehumidifier are examples of *Humidity* ACTUATORS. Similar to the *Temperature SpaceFactor*, the decay of humidity depends on the external environment and the barrier which separates the *System* and the outside environment.

MatterMovement The *MatterMovement SpaceFactor* contains the instantaneous velocity of the matter relative to the *System*. It splits into three components: (1) X-axis *SpaceSubfactor*, (2) Y-axis *SpaceSubfactor*, (3) Z-axis *SpaceSubfactor*. The unit of each component is in meter per second (m/s). This *SpaceFactor* relates closely to *MatterComposition SpaceFactor* because the decay rate depends on the matter type. The *MatterMovement SpaceFactor* may also get affected by the *Temperature SpaceFactor* because hot gas or liquid rises and cold gas or liquid sinks. The examples of *MatterMovement* ACTUATORS are a fan and an air conditioner.

5.3.2 Spatial Resolution

Most technological implementations started by releasing the basic features. Then, they released multiple incremental features and, therefore, gradually fine-tuned the user experience. In its infancy, the current IOT also progresses by aiming the basic human interactions, for example, managing the luminosity of a room. Despite its ubiquity, these interactions are simpler to be modeled and predicted. The construction of its context awareness is still spatially coarse. As it starts to get a grip on the right design, we would expect that more advanced and specific interactions progressively get addressed. The IOT devices would also have more capabilities and precision. This is where the context awareness becomes finer and delicate. The IOT models would work harder to shape a more harmonious cosmos. In this futuristic technological state, the IOT may be capable to engage the right luminosity on the book you are reading OR to add more dose of water only on the withering plants.

Due to this expectation, MESH implements this feature to increase the level of spatial detail in the framework gradually as necessary. Increasing the resolution also enables more precision on the sensor and actuator. We could view the IOT system in more detail and more possible types of interaction can be developed

from there. Consequently, the simulation of the IOT model becomes more complex with more possible high-level applications. However, a higher degree of resolution comes with a cost of computing power and storage. MESH implements the system using a set of 3-dimensional arrays. Doubling the resolution would theoretically increase the need for computing power and storage by $2^3 = 8$.

5.3.3 Function Composition

A *System* is composed of a number of *Entities* which diverse in terms of types and capabilities, for example, a normal light and a smart light. To build an instance of an *Entity* with a set of desired capabilities, MESH utilizes the concept of *Function* composition. A *Entity* may have zero or more *Functions*. An example of an *Entity* that does not have *Functions* is a bookshelf without any intelligent features.

Actuate This *Function* enables the *Entity* to manipulate the state of various *SpaceFactors*. A simulated ACTUATOR defines the *Actuate* method according to the behavior of a real ACTUATOR. As an example, a physical light emits visible light with a specific spectrum profile and also heat to the surrounding. This IOT device can be simulated by defining an *Actuate Function* that changes the *Luminosity SpaceFactor* and the *Temperature SpaceFactor*. This *Function* implementation executes within a *Time Keeper* iteration described in Algorithm 1.

Sense This *Function* enables the *Entity* to detect and measure the current state of various *SpaceFactors*. This *Function* is essential to build a SENSOR. An *Entity* can be designed to have the precision similar to a real device, for example, a motion sensor that has a sight angle of 90°. This IOT device can be simulated by defining a *Sense Function* that detects any changes in the *MatterComposition SpaceFactor*. Similar to *Actuate Function*, this *Function* implementation executes within a *Time Keeper* iteration described in Algorithm 1.

Tasked This *Function* defines that an *Entity* can handle a `task` or be controlled by another *Entity*. Most of the *Entities* have this *Function* defined because *Tasked* handles both system-level `tasks` and entity-level `tasks`. System-level `task` is common for an *Entity* that performs at least a SERVICE.

Table 6: *Tasked* Task Names in MESH

Task Name	Value Format	Description
System-Level		
GET_SYSTEM_INFO	N/A	To get system-level info of an *Entity*
SET_POWER	{"power": obj}	To set input power of an *Entity*
ACTIVE	N/A	To activate an *Entity* in system-level
DEACTIVATE	N/A	To deactivate an *Entity* in system-level
ACTUATE	{"space": obj, "location": obj, "orientation": obj}	To execute *Actuate Function*
SENSE	{"space": obj, "location": obj, "orientation": obj}	To execute *Sense Function*
Entity-Level		
GET_INFO	N/A	To get entity-level info of an *Entity*

Table 7: *Tasked* Task Response in MESH

Status	Result
OK	depends on the `task`
ERROR	{"error": string}

In MESH, we can add a light connected to a light switch. When we flip the light switch, MESH internally sends a system-level `task` to the light to supply the electric power. The system-level `task` is used by MESH to execute the functionality in system level. On the other hand, a smart light is switched on by a user through its mobile device. The simulated mobile device uses an entity-level `task` to encapsulate the request.

A `task` is defined as

$$\texttt{task}(task\_name, value)$$

Table 6 shows the non-exhaustive list of 's task name.

A `task` is always followed by a `task` response which is defined as:

$$\texttt{taskResponse}(status, result)$$

Table 7 lists the possible `statuses` in a `taskResponse`. The `result` content depends on the requested `task`.

Compute This *Function* enables an *Entity* to start a separate computing process parallel to the main process. By utilizing this *Function*, it could simulate a *Entity* that has a computational power for a general-purpose feature, for example, to start a web server, to run an IOT model, or to analyze sensory raw data on the chip.

Powered This *Function* allows an *Entity* to require electric power to be functional. This *Function* is used when we need more realistic simulation in MESH. Concretely, a refrigerator needs electric power to function. Therefore, we need to supply power through a power cable. We can append a *Powered Function* to the refrigerator instance. However, in a less realistic simulation which the IOT model does not concern with this aspect, *Powered* can be omitted and we assume that the refrigerator is always supplied with the power.

5.4 Implementation

We have created a MESH implementation written in Python. The source code is publicly available at `https://github.com/yosefsaputra/Mesh`. When this thesis is being written, MESH development is still in progress. Any changes will be communicated on the GitHub page. In this section, we will discuss how to use MESH in a simple example to uses a *Space* with a power supply, a light switch, and a smart light. First, we look at Table 8 which shows the design of MESH *API*. We choose Python as the development language because of the abundant Python libraries for data science and the object-oriented paradigm. The development of the IOT model is a data analysis and manipulation problem. Therefore, there are great benefits to have access to many data science and machine learning libraries. With object-oriented language, MESH will be extensible for more features, *SpaceFactors*, and *Entities*.

Using MESH in Command-Line Currently, MESH only supports the simulation without any prior saved *System* state. When MESH starts, it always starts from fresh and no *System* is running. MESH uses several Python libraries which can be found in *requirements.txt*. Listing 7 shows several MESH commands in action to create a *System* with a power supply, a light switch, and a smart light. Then, we

Table 8: MESH *API*

Command	**Description**
System Command	
load <system.json>	To load *system.json* to create and to populate a *System*
load <interaction.json>	To load *interaction.json* to populate the interaction bank
create system -n <name>	To create a system with *name*
create space -d <x,y,z>, -r <resolution>	To create a space in the system with dimension of *x,y,z* and resolution of *resolution*
create entity -u <uuid> -t <type> -p <x,y,z> -dx <x_mul,y_mul,z_mul>	To create an entity in the system with UUID of *uuid*, type of *type*, position of *x,y,z*, and dimension multiplier of *x_mul,y_mul,z_mul* in space (spacefactor selection is unsupported)
create channel -t type -s <s_no> -d <d_no>	To create a channel of *type* between entity *s_no* and entity *d_no*
list system	To list and to get details of the current *System*
list space	To list and to get details of the current *Space*
list entity	To list and to get details of the *Entities*
list channel	To list and to get details of the *Channels*
view space	To view the state of spacefactors
Time Manipulation Command	
runstep	To run a *timestep*
run	To run *timesteps* continuously
stop	To stop MESH from running *timesteps* continuously
ct <milliseconds>	To modify the value of *timestep* with *milliseconds* (default=1000ms)
Interaction Command	
task -l <task_level> -t <task_name> -i <entity_no>	To send a `task` with task level of *task_level* task name of *task_name* to *Entity entity_no* (*value* is unsupported)

Listing 7: Using MESH in Command-Line

```
python mesh.py
===== Welcome to Mesh! =====
>> create system -n MySystem

>> create space -d 40,30,12 -r 1

>> create entity -t power_supply -p 0,0,0
>> create entity -t switch -p 19,14,9
>> create entity -u 859e35d2-62c9-4e8d-ab23-7aca3a0f25f4 -t light -p 9,14,9 -dx 1,1,1

>> list entity // returns 0-power_supply, 1-switch, 2-light

>> create channel -t power_wire -s 0 -d 1
>> create channel -t power_wire -s 1 -d 2

>> load system.json // above steps can be replaced with this line
>> load interaction.json // to load interaction .json

>> task -l SYSTEM -t ACTIVATE -i 0 // to activate the power supply
>> task -l SYSTEM -t ACTIVATE -i 1 // to activate the light switch
>> task -l SYSTEM -t ACTIVATE -i 2 // to activate the light

>> runstep
>> view space >> space.dat // to dump spacefactor states to file space.dat
>> task -l SYSTEM -t DEACTIVATE -i 2 // to activate the light

>> run
>> stop

>> quit
```

connect them together with *Channels* so that the electric power can be transferred from the power supply to the light. We start by running the main mesh.py file with python. It starts the running command-line workspace. After the setup, we could interact with the *Entities* by sending system-level `tasks`. Then, we manipulate the simulated time by executing time-manipulation commands.

Programming with MESH On the other hand, we could also use MESH directly in Python. At this point, this method exposes more MESH features and attributes. Listing 8 shows the equivalent Python code with the scenario in the command-line above.

Adding a New *Entity* Adding a new type of *Entity* requires a new class extending *Concrete* class. After overriding several methods, we can add and define *Functions* to the new *Entity*. Listing 9 shows the existing *Light* class with all coded features and *Functions*.

Listing 8: Programming with MESH

```
def main():
    # Init Mesh
    mesh = Mesh()

    # Create System
    system = System('MySystem')

    # Put Space on the System
    system.put_space(dimension=(40, 30, 12), resolution=1,
                     space_factor_types=[i for i in SpaceFactor.SpaceFactor])

    # Put Entity on the Space
    power_supply = PowerSupply(uuid=uuid.uuid4())
    light_switch1 = Switch(uuid=uuid.uuid4())
    wire_ls1 = PowerWire(power_supply, light_switch1)
    light1 = Light(uuid=uuid.uuid4())
    wire_l1 = PowerWire(light_switch1, light1)

    system.put_entity(power_supply, (19, 14, 9))
    system.put_entity(light_switch1, (19, 14, 9))
    system.put_entity(light1, (9, 14, 9))

    mesh.load_system_file("system.json") # above steps can be replaced with this line
    mesh.load_interaction_file("interaction.json") # to load interaction.json

    power_supply.send_task(SystemTask(TaskName.ACTIVE))
    light_switch1.send_task(SystemTask(TaskName.ACTIVE))
    light1.send_task(SystemTask(TaskName.ACTIVE))

    system.runstep()
    print(self.system.space.space_factors)
    light1.send_task(SystemTask(TaskName.DEACTIVATE))

    system.run()
    system.stop()

    light1.destroy() // to clean up the other computing processes
```

Listing 9: Adding a New Entity

```
class Light(Concrete):
    identifier = 'light'
    default_dimension = (3, 3, 3)
    default_orientation = (0, 1, 0)

    default_consume_power_ratings = [ElectricPower(110)]

    default_hue = 0
    default_saturation = 0
    default_brightness = 80

    default_wattage = 15

    default_temperature_raise = 5 # Kelvin

    def __init__(self, uuid, dimension_x=(1, 1, 1),
                 selected_functions=(Function.POWERED, Function.TASKED, Function.COMPUTE,
                     Function.ACTUATE),
                 hue=default_hue, saturation=default_saturation, brightness=default_brightness,
                 wattage=default_wattage,
                 temperature_raise=default_temperature_raise):
        // ...

    def get_default_shape(self):
        // implement the default shape of the new light

    def validate_functions(self, selected_functions):
        // validate the valid set of functions for the new light

    def define_functions(self, selected_functions):
        // define how the functions are implemented, extending the base function
        // Here is an example
        if Function.TASKED in selected_functions:
            self.functions[Function.TASKED] = LightTasked(self)
        if Function.ACTUATE in selected_functions:
            self.functions[Function.ACTUATE] = LightActuate(self)
        // ...

class LightTasked(Tasked):
    // define the supported task names
    tasks = [TaskName.GET_SYSTEM_INFO, TaskName.ACTIVE, TaskName.DEACTIVATE,
        TaskName.GET_INFO]

    def handle(self, task):
        // implement on how to handle each task name
        powered = self.entity.get_function(Function.POWERED)
        power = powered.get_power_input().get_power()
        if task.level == TaskLevel.ENTITY and power in powered.input_power_ratings:
            if task.name == TaskName.GET_INFO:
                task_response = TaskResponse(Status.OK, {'info': get_info()})
            else:
                task_response = TaskResponse(Status.ERROR, {'error': 'Not Implemented'})
        return task_response

// Another Function implementation
class LightActuate(Actuate):
    def __init__(self, entity):
        // ...

    def actuate(self, space, location, orientation):
        // implement how the new light actuate the spacefactors ...
        return self.space_factors
```

Chapter 6

Evaluation

6.1 Warble

In this section, we move from these qualitative judgments toward quantitative ones. We also benchmark the trade-offs associated with these programming simplifications.

6.1.1 Measuring Ease of Programming

To measure ease of programming, we created an application and implemented diverse *Bindings*. We use MetricsReloaded (MetricsReloaded, 2011) to benchmark the implementation. Our experiments use a combination of devices, including a Philips Hue Bridge, Philips Hue Lights, a Wink hub, and GE lights. To control the experiment, we hardcoded locations. All implementations use the same Android communication services.

To characterize the overhead of employing WARBLE *middleware*'s abstractions, we also benchmark the latency and energy costs of interacting with *Things*. We use Android logging to measure latency and the Trepn Profiler (Trepn, 2010) to measure energy consumption using a sampling rate of 10Hz. All of our measurements are done using a Moto X (2nd Gen.) with Android 5.1 Lollipop, which covers 85% of Android devices.

We first compare a native implementation of a *OneTimeBinding* with WARBLE *middleware*'s implementation. In the native implementation, the application finds the closest light by manually querying the Philips Hue Bridge for the available lights, computing their distances to the user's *Controller* location, selecting the closest ones, and performing an HTTP request to the Philips Hue Bridge. In the case of the thermostat, we assume each thermostat exposes its own interface that the *Controller* can interact with. This interface is not limited to HTTP requests but extensible to any communication protocol supported by both systems. Each implementation executes a query to the thermostats and selects the one discovered that is closest to the *Controller*'s location. Subsequently, the application sends

Table 9: WARBLE *middleware*'s code metrics; WARBLE *middleware*'s improvements are dramatic for all three metrics

	One-time		Dynamic	
Metric	Native	WARBLE	Native	WARBLE
Cyclomatic Complexity	3	1	25	1
Max # Indentations	4	2	5	0
Line of Codes	34	13	101	7

commands to the lights and thermostats according to the desired ambient conditions using the appropriate interface given the particular `type` of each *Thing*. In contrast, the application implemented with WARBLE *middleware OneTimeBinding* uses exactly the code in Listing 3 to achieve the same goal.

Table 9 gives the quantitative comparisons between the two versions. We use three programmability metrics: (1) the *maximum number of indentations*, since a high level of nesting reduces code readability Oman and Cook (1988); (2) *cyclomatic complexity* McCabe (1976), which counts paths through the code; (3) and *lines of code*, which is a crude measure of programmer effort. The WARBLE *middleware* implementation of the *OneTimeBinding* is substantially better on these metrics than the native implementation. For instance, the native approach requires more than 2x the number of lines of code than that by WARBLE *middleware*; this difference is even greater if we include *Thing* discovery, especially in the common case that multiple *Adapters* are used. Additionally, WARBLE *middleware*'s abstractions are future-proof towards new types of *Things*, context, and selection strategies, whereas the native implementation only considers hard-coded lights and thermostats.

WARBLE *middleware*'s *DynamicBinding* We use a similar scenario to evaluate the degree to which the *DynamicBinding* simplifies the implementation of continuous interactions with *Things*. In supporting this continuous behavior, the application must make ongoing adjustments in response to a changing set of available *Things*. For instance, as the user's location changes, the user may continuously require a different set of closest lights and thermostats to maintain the desired ambient conditions. The native application must manually adjust bound *Things* and control different sets of *Things* based on the changing context. In contrast, WAR-

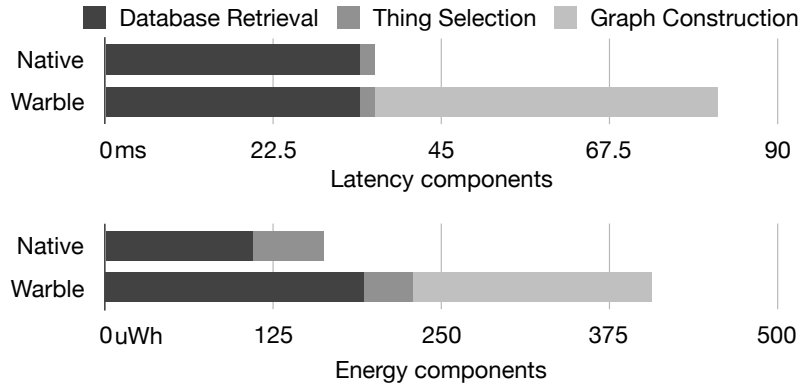

Figure 9: Latency and energy performance components of WARBLE *middleware* vs. a native implementation for retrieving *Things*.

BLE *middleware*'s *DynamicBinding* handles this seamlessly on behalf of the application. The application implemented with WARBLE *middleware* uses the code in Listing 5; Listing 6 shows the analogous native implementation.

Table 9 includes programming metrics for continuous interaction. WARBLE *middleware*'s *DynamicBinding* significantly reduces the programming effort. The level of complexity decreases dramatically because WARBLE *middleware*'s abstractions allow developers to easily and directly capture the user's intent in the binding's `template` and `plan`. WARBLE *middleware* then evaluates and adjusts its fulfillment of the user's needs in the background. This is a typical use case of personalized IoT applications.

6.1.2 The Overhead of WARBLE *middleware*

Simplifying programming using a middleware like WARBLE *middleware* comes at a cost in terms of energy and latency. Figure 9 shows the components of the latency and energy usage for both approaches. Each reported energy and latency measurement is an average of 30 repetitions.The errors for the total energy are $\pm 4.5 \mu Wh$ for the native case and $\pm 19 \mu Wh$ for the Warble case, both of which are within $\pm 5\%$. The magnitudes of the energy costs are dependent on the particular device settings; what is relevant, however, is the relative difference between the two implementations. There are three major contributors to each measurement: retrieving the available *Things* from a database (i.e., registry), constructing the IoT

graph (only in the WARBLE *middleware* case), and *Thing* selection. WARBLE *middleware*'s added latency is entirely in constructing the IoT graph. This graph is used by the *ThingManagerHelper* to carry out the applications' interactions, so the cost is amortized over the *Proxy*'s lifetime. In the native case, in contrast, an application would incur more exceptions in interacting with *Things* behind *Proxies*, having to handle exceptions manually in the application code. The energy difference is also sizable; in addition to the cost of graph construction, WARBLE *middleware* also maintains more detail in the *ThingRegistry* that must be navigated to resolve requests. However, the magnitudes of these latency and energy values are well with reason for today's mobile devices.

6.2 MESH

In this section, we evaluate MESH qualitatively and quantitatively. The qualitative evaluation presents a feature-comparison table among various IOT frameworks. For the quantitative evaluation, a simple setup is used to measure the performance based on different dimensions and resolutions

6.2.1 Qualitative Feature Comparison

We look at several other IOT frameworks that are similar to MESH. Table 10 shows a feature comparison table among the IOT frameworks. This comparison focuses on the features that are useful for achieving the level of detail in the IOT envisioned in Chapter 2. MESH supports most of these features because it is designed to fulfill these requirements. CupCarbon (Bounceur, 2016) focuses on the communication among devices in a smart city setting whereas SimpleIoTSimulator™ (SimpleIoTSimulator) builds a framework for simulating the data collection through sensor networks with different communication protocols. IOTSim (Zeng et al., 2017) simulates the internal computation of an IOT device and how to improve it with situational algorithms, like MapReduce (Dean and Ghemawat, 2008). Despite Gazebo (Koenig and Howard, 2004) is the closest 3D simulation with *SpaceFactors* (used largely in robotics), it lacks realistic device interactions, for example, a light switch transfers power to the attached light. It does not have a separate process for device computational work. MESH still lacks GUI and the support on available

Table 10: MESH vs Other IOT Frameworks

Features	MESH	CupCarbon	SimpleIoT	IOTSim	Gazebo
3D *Space* Simulation	✓	✗	✗	✗	✓
Time Simulation	✓	✓	✓	✗	✓
Actuator Simulation	✓	✗	✗	✗	✓
Sensor Simulation	✓	✗	✓	✗	✓
Interaction Simulation	✓	✗	✗	✗	✓
Spatial Resolution	✓	✗	✗	✗	✗
Connection Simulation	✓	✓	✓	✗	✗
GUI	✗ *separate*	✓	✓	✗	✓
Communication Protocol	✗ *planned*	✓	✓	✗	✗
Device Computation	✓	✗	✗	✓	✓
Level of Detail	high	med	med	low	med

communication protocol in the framework. The GUI itself is planned to be developed separately from MESH whereas the communication protocol support has been planned.

6.2.2 Performance

MESH performance is mainly influenced by the *Space* dimension and its resolution. The manipulation of data in 3D arrays is computational intensive which theoretically increases by the power of 3. Additionally, other factors that linearly affect the performance are:

- the number of *Entities* in the *System*
- the number of *SpaceFactors* in the *Space*
- the number of interactions per *timestep*
- the complexity of internal processes in the *Entities*

To characterize the performance, we take a measurement on the duration for executing one *timestep* in the simulator. The hardware specification is as follows:

- Intel® Core™ i5-2500k CPU @ 3.30 GHz

- Chipset Z68
- RAM 8GB 1600 MHz
- NVIDIA GeForce RTX™ 2060
- Ubuntu 18.04 LTS Bionic Beaver

For all the performance measurements, we create a *System* that comprises of a power supply, two light switches, and two smart lights. The *Space* uses all available *SpaceFactors* and the *Entities* use all supported *Functions*. Both lights have a separate computing process each (utilizing the *Compute Function*) that computes floating-point arithmetic operations continuously in an indefinite loop. This is to simulate a computational-intensive IOT model in a typical MESH usage. Figure 10 shows the *timestep* measurement data for different dimension and resolution. For a given resolution value, the *timestep* duration increases exponentially. Increasing the resolution value increases the *timestep* significantly.

As MESH framework is computationally intensive, we recommend running MESH in a system with multiple processors. When the IOT model involves statistics and machine learning models, we could use Python libraries, like TensorFlow (Abadi et al., 2015), PyTorch (Paszke et al., 2017), or Scikit-learn (Pedregosa et al., 2011). These libraries support GPU utilization to gain better performance and they are compatible with MESH.

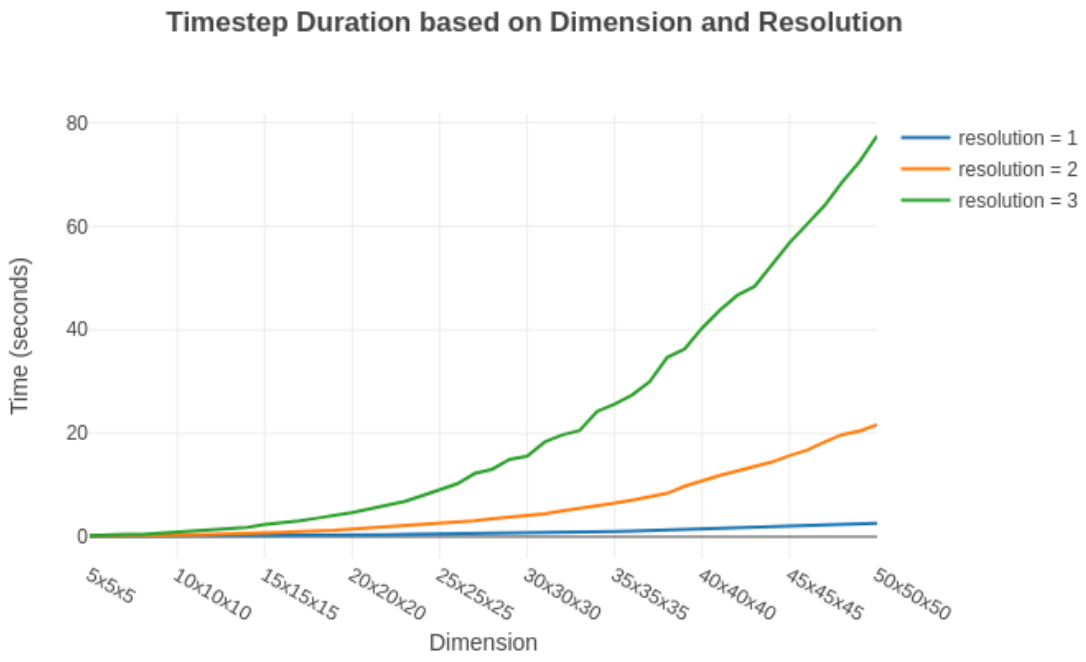

Figure 10: MESH *timestep* Duration based on Dimension and Resolution

Chapter 7

Conclusion

The INTERNET OF THINGS technology is moving forward faster than ever and it is going to revolutionize our lives in different aspects because of its ubiquity. However, there are still challenges in creating the software abstraction to manage the data and other virtual moving parts of billions of IOT devices. Most of the existing implementations are still far to accommodate natural human-device interactions and to embed the new technologies seamlessly into our daily life. Therefore, WARBLE explores the hidden possibilities to establish the pervasiveness of the IOT. In Chapter 2, WARBLE disentangles the complexity of human-device interactions in a non-IOT world and uses the knowledge to build the IOT architecture and the IOT models.

This study also focuses on realizing a user-driven IOT architecture by implementing WARBLE *middleware*. We discuss the internal architecture, the API, and the implementation of WARBLE *middleware*. It has three main aspects of *interoperability*, *personalization*, and *ease of programming*. WARBLE *middleware* encapsulates the device complexities and captures the user interactions. Then, it uses the concept of *Binding* to create interim virtual connections to the relevant devices based on the user context and needs.

WARBLE actively researches on the IOT architecture and models. To facilitate the study, MESH provides the capability to analyze the prototype of the architecture or model in a simulated IOT environment. Currently, MESH targets the interaction that reflects an environmental necessity, like comfortable room temperature. In this thesis, we discuss the conceptual architecture and the implementation of MESH.

Future Work In order to establish the ideal IOT, there is a large gap in the IOT architecture and models to handle the management of IOT components and to bring the IOT services to the users naturally. The architecture defines the infrastructure, the components, and the interactions among the components in the IOT technology. An effective architecture allows the components aware of their roles and functionalities, therefore, they are able to understand each other supporting

the IOT technology together. The architecture must be scalable and reliable. Additionally, security and privacy are two essential qualities that are inseparable with any IOT applications.

The IOT will generate a colossal amount of data from the human-device interactions. Manual data analysis does not scale in this situation. Therefore, our knowledge of data science and machine learning will definitely help to analyze the data and to turn it into a set of abstract patterns. Subsequently, the IOT models are built to represent these patterns. Research in this area enables the formulation of the appropriate methodologies to perform this analysis.

On the other hand, a research topic on abstracting human-device interactions will improve the user experience when interacting with the IOT technology. Natural interactions are necessary so that the users do not realize that they interact with technology. Nowadays, flipping a light switch to turn on a light is a natural and mundane action whereas this convenience amazed many people back then.

Specifically for the WARBLE *middleware*, its current state lacks support on more IOT ecosystems and also advanced *Selectors*. More extensive evaluation of the user-driven architecture is also important to compare with other architectures. For MESH, improving the performance is the priority while creating GUI would be very useful to visualize the *System*. Non-spatial properties, which express the user intents and needs, are necessary to make MESH more comprehensive to be an IOT framework.

Bibliography

Hypercat helps smart-home gadgets get along. *PC Pro*, (252):124–125, 10 2015. URL http://ezproxy.lib.utexas.edu/login?url=https://search-proquest-com.ezproxy.lib.utexas.edu/docview/1711126936?accountid=7118. Copyright - Copyright Dennis Publishing Ltd. Oct 2015; Document feature - Photographs; Last updated - 2016-08-27.

Nfc forum; definitive internet of things and nfc white paper published by nfc forum, Jul 12 2016. URL http://ezproxy.lib.utexas.edu/login?url=https://search-proquest-com.ezproxy.lib.utexas.edu/docview/1802259510?accountid=7118. Name - Google Inc; Copyright - Copyright 2016, NewsRx LLC; Last updated - 2016-07-07.

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL http://tensorflow.org/. Software available from tensorflow.org.

Jairo Ariza, Camilo Mendoza, Kelly GarcÃl's, and NicolÃąs Cardozo. A research agenda for iot adaptive architectures. *Proceedings*, 2(19):1229, Oct 2018. ISSN 2504-3900. doi:10.3390/proceedings2191229. URL http://dx.doi.org/10.3390/proceedings2191229.

Kent Beck. Design patterns: Elements of reusable object-oriented software. *IBM Systems Journal*, 34(3):544–545, 1995. URL http://ezproxy.lib.utexas.edu/login?url=https://search.proquest.com/docview/222413156?accountid=7118. Copyright - Copyright International Business Machines Corporation 1995; Last updated - 2014-04-08; CODEN - IBMSA7; SubjectsTermNotLitGenreText - US.

Ahcène Bounceur. Cupcarbon: A new platform for designing and simulating smart-city and iot wireless sensor networks (sci-wsn). In *Proceedings of the International Conference on Internet of Things and Cloud Computing*, ICC '16, pages 1:1–1:1, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4063-2. doi:10.1145/2896387.2900336. URL http://doi.acm.org.ezproxy.lib.utexas.edu/10.1145/2896387.2900336.

Jollen Chen. Devify: Decentralized internet of things software framework for a peer-to-peer and interoperable iot device. *SIGBED Rev.*, 15(2):31–36, June 2018. ISSN 1551-3688. doi:10.1145/3231535.3231539. URL http://doi.acm.org/10.1145/3231535.3231539.

S. Cheshire and M. Krochmal. Multicast dns. RFC 6762, RFC Editor, February 2013. URL http://www.rfc-editor.org/rfc/rfc6762.txt. http://www.rfc-editor.org/rfc/rfc6762.txt.

J. D. Day and H. Zimmermann. The osi reference model. *Proceedings of the IEEE*, 71(12):1334–1340, Dec 1983. ISSN 0018-9219. doi:10.1109/PROC.1983.12775.

Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008. ISSN 0001-0782. doi:10.1145/1327452.1327492. URL http://doi.acm.org/10.1145/1327452.1327492.

Markus Eisenhauer, Peter Rosengren, and Pablo Antolin. A development platform for integrating wireless devices and sensors into ambient intelligence systems. In *Sensor, Mesh and Ad Hoc Communications and Networks Workshops, 2009. SECON Workshops' 09. 6th Annual IEEE Communications Society Conference on*, pages 1–3. IEEE, 2009.

J. F. Ensworth and M. S. Reynolds. Ble-backscatter: Ultralow-power iot nodes compatible with bluetooth 4.0 low energy (ble) smartphones and tablets. *IEEE Transactions on Microwave Theory and Techniques*, 65(9):3360–3368, Sep. 2017. ISSN 0018-9480. doi:10.1109/TMTT.2017.2687866.

Dominique Guinard and Vlad Trifa. Towards the web of things: Web mashups for embedded devices. In *Workshop on Mashups, Enterprise Mashups and Lightweight*

Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences), Madrid, Spain, April 2009.

D. Hardt. The oauth 2.0 authorization framework. RFC 6749, RFC Editor, October 2012. URL http://www.rfc-editor.org/rfc/rfc6749.txt. http://www.rfc-editor.org/rfc/rfc6749.txt.

Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1998. ISBN 0132733501.

IFTTT, 2011. https://ifttt.com/, 2011.

JSON, 1999. https://www.json.org/, 1999.

Park Jun-Hong, Kim Hyeong-Su, and Kim Won-Tae. Dm-mqtt: An efficient mqtt based on sdn multicast for massive iot communications. *Sensors*, 18(9), 09 2018. URL http://ezproxy.lib.utexas.edu/login?url=https://search-proquest-com.ezproxy.lib.utexas.edu/docview/2126876470?accountid=7118. Copyright - Âl' 2018. This work is licensed under http://creativecommons.org/licenses/by/4.0/ (the âĂIJLicenseâĂİ). Notwithstanding the ProQuest Terms and Conditions, you may use this content in accordance with the terms of the License; Last updated - 2018-11-17; SubjectsTermNotLitGenreText - South Korea; United States–US.

N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 3, pages 2149–2154 vol.3, Sep. 2004. doi:10.1109/IROS.2004.1389727.

D. Lan, Z. Pang, C. Fischione, Y. Liu, A. Taherkordi, and F. Eliassen. Latency analysis of wireless networks for proximity services in smart home and building automation: The case of thread. *IEEE Access*, 7:4856–4867, 2019. ISSN 2169-3536. doi:10.1109/ACCESS.2018.2888939.

T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2 (4):308–320, December 1976.

MetricsReloaded, 2011. https://plugins.jetbrains.com/plugin/93, 2011.

Nest, 2011. https://nest.com, 2011.

E. D. Ngangue Ndih and S. Cherkaoui. On enhancing technology coexistence in the iot era: Zigbee and 802.11 case. *IEEE Access*, 4:1835–1844, 2016. ISSN 2169-3536. doi:10.1109/ACCESS.2016.2553150.

Anne H Ngu, Mario Gutierrez, Vangelis Metsis, Surya Nepal, and Quan Z Sheng. Iot middleware: A survey on issues and enabling technologies. *IEEE Internet of Things Journal*, 4(1):1–20, 2017.

Node-RED, 2013. https://nodered.org/, 2013.

P.W. Oman and C.R. Cook. A paradigm for programming style research. *ACM Sigplan Notices*, 23(12):69–78, 1988.

OpenSense, 2010. http://www.opensense.ethz.ch/trac/, 2010.

OWL, 2012. Web Ontology Language (OWL). https://www.w3.org/OWL/, 2012.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python . *Journal of Machine Learning Research*, 12:2825–2830, 2011.

V. Pehkonen and J. Koivisto. Secure universal plug and play network. In *2010 Sixth International Conference on Information Assurance and Security*, pages 11–14, Aug 2010. doi:10.1109/ISIAS.2010.5604189.

Per Persson and Ola Angelsmark. Calvin–merging cloud and iot. *Procedia Computer Science*, 52:210–217, 2015.

Philips Hue, 2012. https://www2.meethue.com, 2012.

Philips Hue API, 2017. https://developers.meethue.com/develop/hue-api/, 2017.

S. R. Pokhrel and C. Williamson. Modeling compound tcp over wifi for iot. *IEEE/ACM Transactions on Networking*, 26(2):864–878, April 2018. ISSN 1063-6692. doi:10.1109/TNET.2018.2806352.

Eko Sakti Pramukantoro and Husnul Anwari. An event-based middleware for syntactical interoperability in internet of things. *International Journal of Electrical and Computer Engineering (IJECE)*, 8:3784 – 3792, 2018. ISSN 2088-8708. doi:http://doi.org/10.11591/ijece.v8i5.pp3784-3792. URL https://www.iaescore.com/journals/index.php/IJECE/article/view/9289.

Yosef Saputra, Jie Hua, Nathaniel Wendt, Christine Julien, and Gruia-Catalin Roman. Warble: Programming abstractions for personalizing interactions in the internet of things. *MOBILESoft*, 6, 2019.

Neetesh Saxena, Santiago Grijalva, and Narendra S. Chaudhari. Authentication protocol for an iot-enabled lte network. *ACM Trans. Internet Technol.*, 16(4):25:1–25:20, December 2016. ISSN 1533-5399. doi:10.1145/2981547. URL http://doi.acm.org.ezproxy.lib.utexas.edu/10.1145/2981547.

SensorML, 2007. sensor model language (sensorml), 2007.

SimpleIoTSimulator. https://www.smplsft.com/SimpleIoTSimulator.html, 2015.

Alessandro Sivieri, Luca Mottola, and Gianpaolo Cugola. Building internet of things software with eliot. *Computer Communications*, 89-90:141 – 153, 2016. ISSN 0140-3664. doi:https://doi.org/10.1016/j.comcom.2016.02.004. URL http://www.sciencedirect.com/science/article/pii/S0140366416300238. Internet of Things $Researchchallengesand Solutions$.

John Soldatos, Nikos Kefalakis, Manfred Hauswirth, Martin Serrano, Jean-Paul Calbimonte, Mehdi Riahi, Karl Aberer, Prem Prakash Jayaraman, Arkady B. Zaslavsky, Ivana Podnar Zarko, Lea Skorin-Kapov, and Reinhard Herzog. Openiot: Open source internet-of-things in the cloud. In *OpenIoT@SoftCOM*, 2014.

John Soldatos, Nikos Kefalakis, Manfred Hauswirth, Martin Serrano, Jean-Paul Calbimonte, Mehdi Riahi, Karl Aberer, Prem Prakash Jayaraman, Arkady Zaslavsky, Ivana Podnar Žarko, et al. Openiot: Open source internet-of-things in the cloud.

In *Interoperability and open-source solutions for the internet of things*, pages 13–25. Springer, 2015.

C.C. Tan, B. Sheng, H. Wang, and Q. Li. Microsearch: A search engine for embedded devices used in pervasive computing. *ACM Transactions on Embedded Computing Systems*, 9(4), April 2010.

ThingSpeak, 2010. http://www.thingspeak.com, 2010.

Trepn, 2010. https://developer.qualcomm.com/software/trepn-power-profiler, 2010.

H. Wang, C. C. Tan, and Q. Li. Snoogle: A search engine for pervasive environments. *IEEE Transactions on Parallel and Distributed Systems*, 21(8):1188–1202, August 2010.

Roy Want. The physical web. In *Proceedings of the 2015 Workshop on IoT Challenges in Mobile and Industrial Systems*, IoT-Sys '15, pages 1–1, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3502-7. doi:10.1145/2753476.2753496. URL http://doi.acm.org/10.1145/2753476.2753496.

Mark Weiser. The computer for the 21st century. *SIGMOBILE Mob. Comput. Commun. Rev.*, 3(3):3–11, July 1999. ISSN 1559-1662. doi:10.1145/329124.329126. URL http://doi.acm.org/10.1145/329124.329126.

Wink, 2014. https://www.wink.com, 2014.

Wink API, 2017. https://winkapiv2.docs.apiary.io/#, 2017.

Withings, 2009. https://www.withings.com, 2009.

J. Wu and J. Dong. A simple service discovery and configuration protocol for embedded devices. In *2006 International Conference on Communication Technology*, pages 1–3, Nov 2006. doi:10.1109/ICCT.2006.341816.

Xively, 2013. http://www.xively.com, 2013.

K.-K. Yap, V. Srinivasan, and M. Motani. Max: Wide area human-centric search of the physical world. *ACM Transactions on Sensor Networks*, 4(4), September 2008.

Kumar Yelamarthi, Md Sayedul Aman, and Ahmed Abdelgawad. An Application-Driven Modular IoT Architecture. *Wireless Communications and Mobile Computing*, 2017:16, 2017. URL 10.1155/2017/1350929.

Xuezhi Zeng, Saurabh Kumar Garg, Peter Strazdins, Prem Prakash Jayaraman, Dimitrios Georgakopoulos, and Rajiv Ranjan. Iotsim: A simulator for analysing iot applications. *Journal of Systems Architecture*, 72:93 – 107, 2017. ISSN 1383-7621. doi:https://doi.org/10.1016/j.sysarc.2016.06.008. URL http://www.sciencedirect.com/science/article/pii/S1383762116300662. Design Automation for Embedded Ubiquitous Computing Systems.

Vita

Yosef Saputra was from Jakarta, Indonesia. After completing his high school in Indonesia, he went to college at Nanyang Technological University in Singapore and graduated with Bachelor Degree in Electrical and Electronic Engineering with Minor in Business. Subsequently, he worked in Micron Technology, Singapore, for 6 years as a Product Engineer. After that, he entered Graduate School in The University of Texas at Austin in Electronic and Computer Engineering, focusing on Software Engineering. During his study, he was actively doing research pertaining to the Internet of Things. He was also a teaching assistant for several software courses.

Address: yosef.saputra@utexas.edu

This thesis was typed by the author.